НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

«На правах рукопису»

УДК _____

«До захисту допущено»

В.о. завідувача кафедри

_____ М.В.Грайворонський

“___”_____2018 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності: 125 Кібербезпека

на тему: _____
_____
_____

Виконав (-ла): студент (-ка) ____ курсу, групи _____
(шифр групи)

Конорєв Олександр Юрійович _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник к.т.н., доц. Родіонов Андрій Миколайович
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензентк.т.н., доцент ХНУРЕ Сінельнікова О.І.

_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою
Спеціальність (спеціалізація) – 125 Кібербезпека («Системи і технології кібербезпеки)

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ М.В.Грайворонський
(підпис)
«___»_____2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Конорєву Олександру Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації: Удосконалення захищеності ядра Linux на платформі ARM,

науковий керівник дисертації к.т.н., доц. Родіонов Андрій Миколайович,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «15» листопада 2018 р. № 4171-с

2. Термін подання студентом дисертації 12.12.2018

3. Об'єкт дослідження _____
_____

4. Предмет дослідження _____
_____
_____
_____

5. Перелік завдань, які потрібно розробити _____
_____
_____
_____
_____
_____

6. Орієнтовний перелік ілюстративного матеріалу _____
_____

_____

7. Орієнтовний перелік публікацій _____

_____

_____

8. Консультанти розділів дисертації*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка

Студент _____ _____
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ _____
(підпис) (ініціали, прізвище)

* Консультантом не може бути зазначено наукового керівника магістерської дисертації.

РЕФЕРАТ

Робота обсягом 100 сторінок містить 62 ілюстрацій, 22 таблиці та 29 літературних посилань.

Метою даної кваліфікаційної роботи посилення безпеки ядра Linux-подібної операційної системи сучасного смартфона або одноплатного комп’ютера, це дасть можливість обмежити роботу руткітів або взагалі не дозволити їм бути запущеними.

Об’єктом дослідження є системи та процеси захисту інформації.

Предметом дослідження є сучасні смартфони та одноплатні комп’ютери.

Результати роботи викладені вигляді методології розробки динамічного аналізатору, PoC пропонованої методології.

Результати роботи можуть бути застосовані у аналізі зловмисного програмного забезпечення на мобільних пристроях, моніторингу, аудиту, логуванню застосунків на мобільних пристроях.

ІНФОРМАЦІЙНА БЕЗПЕКА МОБІЛЬНИХ ПРИСТРОЇВ, ДОВІРЕНЕ СЕРЕДОВИЩЕ ВИКОНАННЯ, ДОВІРЕНИЙ ЗАСТОСУНОК, БЕЗПЕКА ІНФОРМАЦІЙНИХ І КОМУНІКАЦІЙНИХ СИСТЕМ

РЕФЕРАТ

Работа объемом 100 страниц содержит 62 иллюстраций, 22 таблицы и 29 литературных источников.

Целью данной квалификационной работы является усиление безопасности ядра Linux-подобной операционной системы современного смартфона или одноплатного компьютера, это даст возможность ограничить работу руткитов или вообще не позволить им быть запущенными.

Объектом исследования являются системы и процессы защиты информации.

Предметом исследования являются современные смартфоны и одноплатные компьютеры.

Результаты работы изложены в виде методологии разработки динамического анализатора, PoC данной методологии.

Результаты работы могут быть применены в анализе вредоносных программ на мобильных устройствах, мониторинга, аудита, логирования приложений на мобильных устройствах

ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ МОБИЛЬНЫХ УСТРОЙСТВ, ДОВЕРЕННАЯ СРЕДА ИСПОЛНЕНИЯ, ДОВЕРЕННОЕ ПРИЛОЖЕНИЕ, БЕЗОПАСНОСТЬ ИНФОРМАЦИОННЫХ И КОММУНИКАЦИОННЫХ СИСТЕМ

ABSTRACT

The work includes 100 pages, 62 figures, 22 tables and 29 literary references.

The aim of this qualification is to strengthen the security of the Linux-like operating system kernel of a modern smartphone or single-board computer, this will make it possible to limit the work of rootkits or not allow them to be started at all.

The object of researches are systems and processes of information security.

The subject of research are modern smartphones and single-board computers.

The results of the work are presented as a method of implementation such a dynamic analysis tool, PoC of this method.

The results of the work can be used in analyzes of malicious software on mobile phones and single-board computed, monitoring, auditing, logging of mobile applications.

MOBILE SECURITY, TRUSTED EXECUTION ENVIRONMENT, TRUSTED APPLICATION, SECURITY OF INFORMATION AND COMMUNICATION SYSTEMS

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ,..... 8

СКОРОЧЕНЬ І ТЕРМІНІВ 8

ВСТУП..... 9

1 ОГЛЯД ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX, ОСОБЛИВОСТІ ARM АРХІТЕКТУРИ, ОСНОВНІ ВИЗНАЧЕННЯ 12

1.1 Linux-подібна операційна система 12

1.2 LKM..... 16

1.3 Принцип роботи load_module..... 21

1.4 MMU 24

1.5 Стек виконання 27

1.6 Технології захисту в Linux..... 29

1.7 Особливості ARM архітектури 30

1.8 ATF, ARM syscall convention,..... 35

1.9 OP-TEE, QEMU..... 37

Висновки до розділу 1 39

2 ОГЛЯД ЗАГРОЗ ТА АНАЛІЗ ІСНУЮЧИХ МЕХАНІЗМІВ ЗАХИСТУ 41

2.1 Типові вразливості та методи захисту в user mode..... 41

2.2 Типові вразливості в privileged mode 52

2.3 Rootkit 57

2.4 Аналіз задачі виявлення руткітів. 62

2.5 Огляд існуючих рішень..... 64

2.6 Цілі посилення захисту 67

Висновки до розділу 2 70

3 ПОСИЛЕННЯ ЗАХИСТУ ЯДРА LINUX 72

3.1 Розробка техніки посилення захисту ядра Linux від rootkit 72

3.2 Аналіз результатів 79

Висновки до розділу 3 81

4 РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ 83

4.1. Опис ідеї проекту 83

4.2. Технологічний аудит ідеї проекту 85

4.3. Аналіз ринкових можливостей запуску стартап-проекту 85

4.4. Розроблення ринкової стратегії проекту 89

4.5. Розроблення маркетингової програми стартап-проекту 90

Висновки до розділу 4 93

ВИСНОВКИ 94

ПЕРЕЛІК ПОСИЛАНЬ 97

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Trusted Execution Environment, TEE, swd [1] – довірене середовище виконання.
Rich Execution Environment, REE, nwd – Звичайна операційна система, наприклад Windows, Ubuntu, Android, iOS.
Атака — детально підібраний набір дій, які, в разі успіху, призведуть компрометації конфіденціальної, цілісності або доступності інформаційних ресурсів.
Поверхня атаки - термін, який застосовується при вирішенні задач інформаційної безпеки комп'ютерних систем і позначає загальну кількість можливих вразливих місць інформаційної системи.
Загроза - будь-які обставини або події, що можуть бути причиною порушення політики безпеки інформації і/або нанесення збитків автоматизованій системі.
Модель загроз - це модель, яка описує можливі загрози інформаційній безпеці ресурсам системи.
Native-застосунок – застосунок розроблений на мові програмування C.
Trustlet – довірена програма (англ. Trusted application, trustlet).

ВСТУП

Сучасні мобільні пристрої стають все більш складними: смартфони обладнують продуктивнішими процесорами і відео чіпами, і вони вже можуть змагатися з повноцінними персональними комп’ютерами. Сучасний смартфон або одноплатний комп’ютер (SoC) вже може підтримувати повноцінні операційні системи і проблеми безпеки інформації, які притаманні звичайним стаціонарним персональним комп'ютерам, знаходять місце і в цій галузі.

Ми живемо у все більш мобільному світі. Згідно з останніми дослідженнями[29] три чверті компаній дозволяють працівникам використовувати персональні пристрої для роботи. Отже працівники можуть зберігати конфіденційні дані на них, та їх втрата або оприлюднення може нашкодити як і самим користувачам так і компанії.

Користувач може загрузити застосунок із недовіреного джерела і зовсім не підозрювати, що цей застосунок може бути зловмисним троянським застосунком, який використовує вразливості операційної системи смартфону для, наприклад підвищення привілеїв, а потім краде, змінює конфіденційні дані.

Серед типових загроз, які здійснюються на практиці зловмисниками - злам смартфона, використовуючи віддалений зв'язок, стеження за діяльністю власника смартфона, а також руткіти й звичайні віруси.

Для знаходження вірусів та руткітів на стаціонарних комп’ютерах зазвичай використовують антивірусні програми або інтроспекцію віртуальних машин, але сучасні віруси можуть дізнатися про існування в системі аналізуючої програми й зупинити свою діяльність, а технологія віртуалізації не настільки розвинута в сфері мобільних пристроїв, щоб її можна було ефективно використовувати. Більш того, самі антивірусні програми можуть мати вразливості, які може використовувати вірусна програма, оскільки вони працюють на одному рівні.

Технологія віртуалізації з апаратною підтримкою[1] може покращити прозорість систем віртуалізації і її ефективність на мобільний пристроях; однак цей підхід все ще залишає артефакти, які можуть бути легко виявлені

зловмисним програмним забезпеченням [4, 5, 6].

Тому в даній роботі пропонується інший метод знаходження шкідливого програмного забезпечення в системі на мобільному пристрої, використовуючи платформо-специфічні особливості ARM пристроїв.

Актуальність роботи зумовлюється тим, що область безпеки мобільних пристроїв швидко розвивається, а разом з цим і розробники зловмисного програмного забезпечення. Притаманні персональним комп'ютерам методи захисту не зовсім ефективно працюють на мобільних пристроях. Представлена робота пропонує метод захисту інформації від руткітів, який використовує особливості ARM пристроїв – довірене середовище виконання (Trusted Execution Environment, trustzone)[1], а також не залишає артефактів, які дають змогу зловмисному програмному забезпеченню виявити віртуальне середовище.

Метою роботи є посилення безпеки ядра Linux операційної системи сучасного смартфона або одноплатного комп'ютера, це дасть можливість обмежити роботу руткітів або взагалі не дозволити їм бути запущеними.

Завданням роботи є:

1) Огляд та аналіз існуючих вразливостей у сфері інформаційної безпеки мобільних пристроїв.
2) Аналіз існуючих методів захисту в Linux подібній операційній системі.
3) Розроблення методу динамічного аналізу завантажуваних модулів.
4) Розробка тестового застосунку, що втілює п.4.

Об'єктом дослідження є системи та процеси захисту інформації.

Предметом дослідження є сучасні смартфони та одноплатні комп'ютери.

Новизна роботи зумовлюється тим, що в роботі запропоновано метод визначення зловмисного програмного забезпечення в операційній системі мобільного пристрою, який відрізняється використанням платформною особливістю ARM пристроїв - Trusted Execution Environment.

Практичне значення полягає в тому, що результати роботи можуть бути застосовані на практиці в реальній операційній системі для покращення її безпеки. Пропоноване рішення може використовуватися в якості динамічного

аналізатора зловмисного програмного забезпечення. Також базі пропонованого рішення можна впровадити політики безпеки виконання завантажуємих модулів тощо.

1 ОГЛЯД ОПЕРАЦІЙНОЇ СИСТЕМИ LINUX, ОСОБЛИВОСТІ ARM АРХІТЕКТУРИ, ОСНОВНІ ВИЗНАЧЕННЯ

1.1 Linux-подібна операційна система

Сучасний смартфон все більш стає схожим на персональний комп’ютер, як за пропонованим функціоналом так і за швидкістю роботи. Як було наведено раніше, сьогоденний смартфон має повноцінну операційну систему, зазвичай це Android або iOS. Також для одноплатних систем може використовуватися Linux дистрибутив.

Для початку треба визначити, навіщо потрібна операційна система. Існує багато прикладів, коли електронні пристрої працюють не опираючись на неї, наприклад, вбудовані системи із інтернету речей. Як для кінцевого користувача, операційна система є графічним інтерфейсом, а також надає можливість користуватися улюбленими застосунками, але це з боку користувача, з іншого боку – розробника програмного забезпечення, операційна система це певний прошарок між апаратною частиною пристрою, та програмним забезпеченням (Рисунок 1.1).

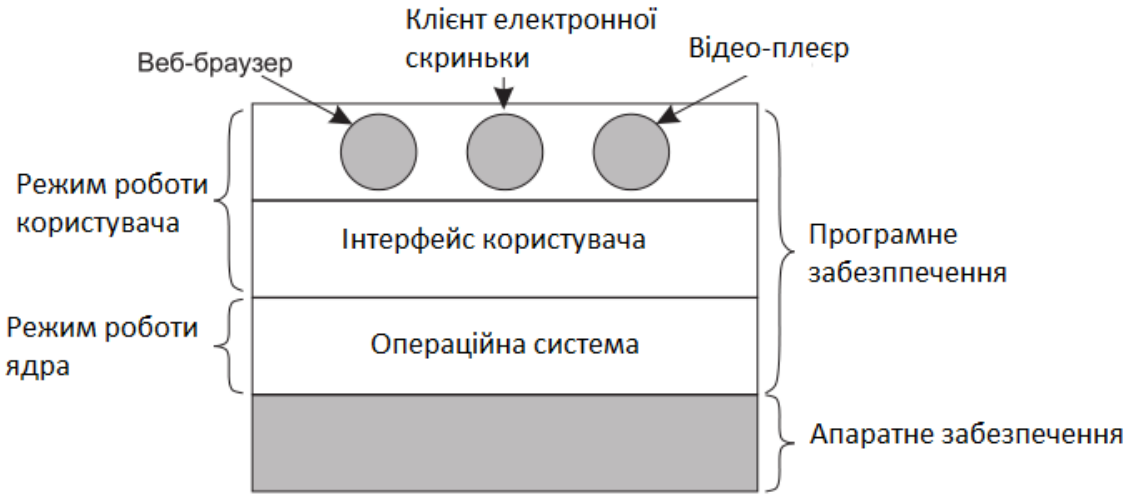

Рисунок 1.1 - Операційна системи в структурі програмного забезпечення

Архітектура більшості комп'ютерів на рівні машинної мови дуже примітивна

і незручна для використання в програмах, особливо це стосується систем введення-виведення. Зрозуміло, що жоден програміст не захоче мати справи з жорстким диском на апаратному рівні. Замість нього обладнанням займається та частина програмного забезпечення, яка називається драйвером диска і надає, не вдаючись у деталі, інтерфейс для читання і запису дискових блоків. Операційні системи містять безліч драйверів для управління пристроями введення-виведення, це значно полегшує роботу розробника, і це лише один з прикладів користі операційної системи.

Варто відзначити, що операційної система не є звичайним програмним забезпеченням, яке працює в режимі користувача, наприклад, якщо користувач не задоволений роботою свого браузера, то він може вибрати іншу програму або, якщо захоче, написати власний браузер, але він не може написати власний обробник переривань системних годин, що є частиною операційної системи і захищений на апаратному рівні від будь-яких спроб внесення змін з боку користувача. Ця різниця іноді не настільки чітко виражено у вбудованих системах (які можуть не мати режиму ядра).

Існують також програми, що можуть виконуватися у просторі користувача, але надають особливий функціонал. Наприклад, досить часто зустрічаються програми, що дозволяють користувачам змінювати їх паролі. Ці застосунки можуть не відноситися до ядру системи, але вони виконують важливу функцію і повинні відповідно бути захищеними. Всі програми, що виконуються у просторі ядра, безумовно, відносяться до операційної системи, але існують застосунки, що виконуються в просторі користувача і мають з нею тісний зв'язок з простором ядру.

Зазначимо декілька важливих функцій операційної системи:

- Управління апаратними засобами, пристроями введення-виведення;
- Файлова система;
- Підтримка багатозадачності;
- Підтримка роботи з інтернетом;

- Робота за пам’яттю;
- та інші…

Розглянемо Linux-подібну операційну систему[22]. Основною її частиною є ядро, воно і надає основні функції. Типовими складовими ядра є:

- Обробники переривань. Він оброблює запити на переривання виконання, що надходять від різних пристроїв;
- Планувальник. Він розподіляє процесорний час між процесами.
- MMU. Ця система відповідає за адресний простір процесів, віртуальну пам'ять тощо;
- Networking
- IPC

Ядра операційних систем можна поділити на такі групи: які використовують монолітне ядро та ті, що використовують мікроядро.

Монолітне ядро імплементовано у якості одного процесу, і він працює в одному адресному просторі. Впроваджується як один статичний бінарний файл (англ. image). Сервіси ядра знаходяться і працюють також в одному адресному просторі. Більшість систем Unix мають монолітне ядро.

Мікроядро[22] – обмежується мінімальною імплементацією низькорівневого функціоналу ядра, наприклад:

- системні виклики;
- управління пам'яттю;
- управління процесами і потоками;
- IPC.

Решту реалізують, так звані, сервіси, які знаходяться у користувацькому адресному просторі. Прикладами сервісів є мережеві сервіси, файлова система тощо. Така конструкція дозволяє збільшити загальну швидкодію системи.

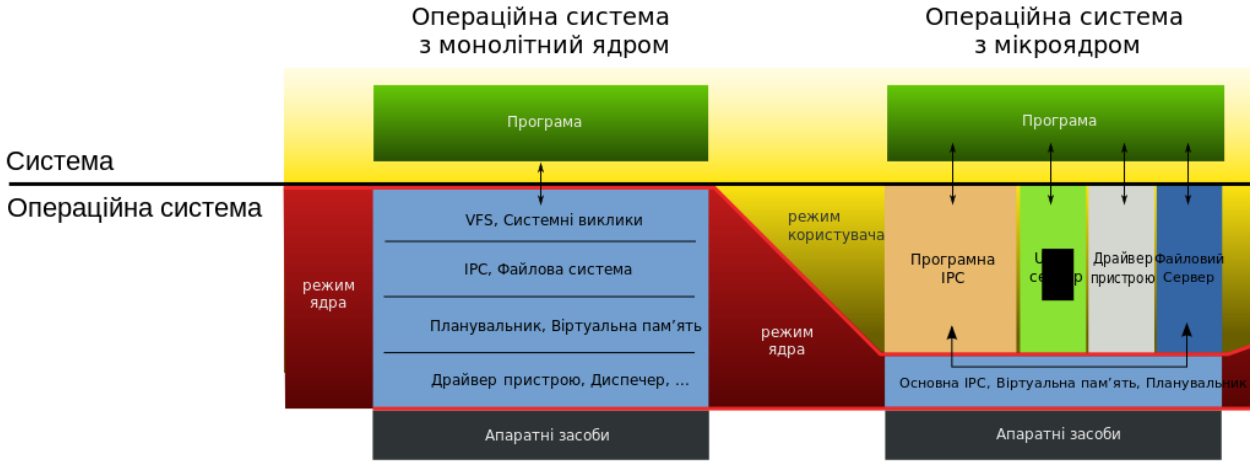

Рисунок 1.2 - Структура операційних систем на монолітному ядрі, мікроядрі.

Також поділ сервісів запобігає можливості падіння одного сервісу при виході з ладу іншого, а ще модульний принцип побудови системи дозволяє вивантажити з пам'яті сервіс, якщо це необхідно другому сервісу.

Ядро ОС Linux монолітне, воно запозичило дещо з мікроядерной архітектури: використання модульного принципу побудови, можливість пріоритетного планування самого себе (kernel preemption), а також можливо динамічного завантажити в ядро зовнішніх модулів ядра(драйверів). В ядрі Linux не використовує функцій мікроядерной моделі, які призводять до зниження продуктивності: все виконується в режимі ядра з безпосереднім викликом функцій замість передачі повідомлень. Отже, операційна система Linux - модульна, багатопоточна, з пріоритетним плануванням самого ядра.

Однією з важливих особливостей Linux є можливість розширити під час виконання набір функцій що пропонується ядром. Це означає, що ви можете додати функціональність до ядра (а також видалити), поки система працює, тобто для цього не потрібно компілювати ядро. Кожен шматочок коду, який можна додати до ядра під час виконання, називається модулем. Ядро Linux підтримує безліч різних типів модулів, включаючи драйвери пристроїв. Кожен модуль складається з об'єктного коду, який може бути динамічно пов'язаний з працюючим ядром за допомогою програми insmod і може бути від’єднано програмою rmmod. На рисунку 1.3 показані різні типи модулів, що відповідають

за конкретні задачі.

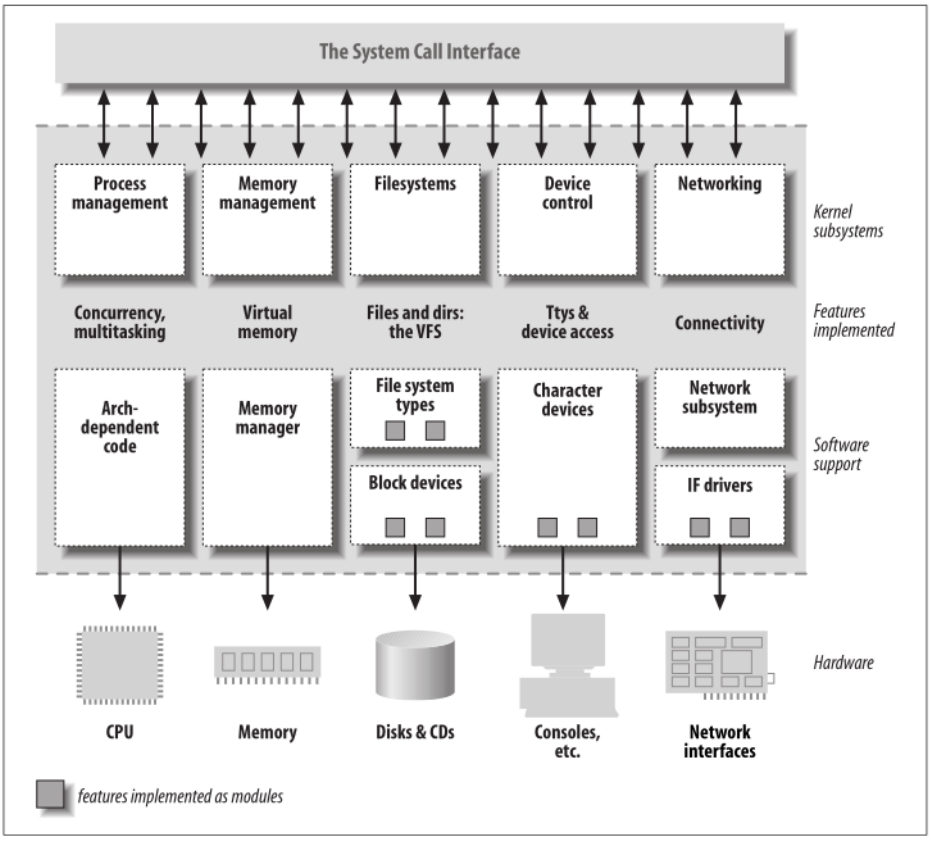

Рисунок 1.3 – Частини ядра Linux.

Варто зазначити, що kernel module виконуються у просторі ядра на відміну від звичайних користувацьких застосунків.

1.2 LKM

Модулі ядра дозволяються додавати драйвери пристроїв, файлових систем та інших компонентів динамічно в ядро Linux без перезавантаження та рекомпіляції системи.

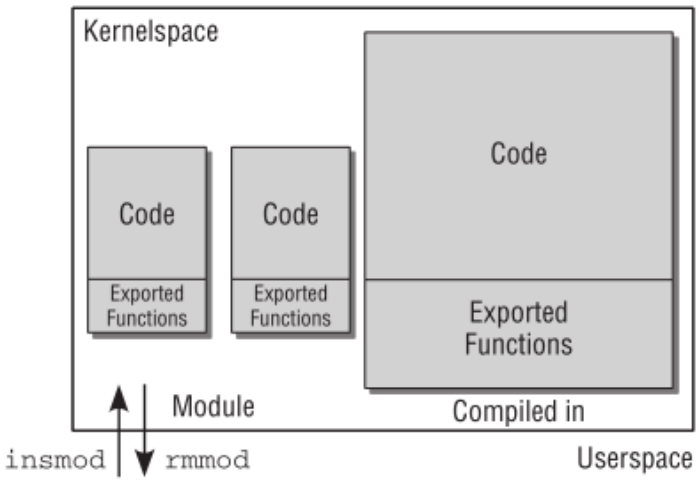

Рисунок 1.4 – Ядро Linux із завантаженим модулем.

Модуль ядра визначається набором структур даних, назва найважливішої структури - module; екземпляр цієї структури виділяється для кожного модуля, завантаженого в ядро. Ця структура досить велика, роздивимось кілька її важливих для нас частин (рисунок 1.4).

```
struct module {
        enum module_state state;

        /* Member of list of modules */
        struct list_head list;

        /* Unique handle for this module */
        char name[MODULE_NAME_LEN];

        /* Sysfs stuff. */
        struct module_kobject mkobj;
        struct module_attribute *modinfo_attrs;
        const char *version;
        const char *srcversion;
        struct kobject *holders_dir;

        /* Exported symbols */
        const struct kernel_symbol *syms;
        const s32 *crcs;
        unsigned int num_syms;
```

Рисунок 1.5 – Частина структури module.

- list - є стандартним елементом списку, який використовується ядром, щоб тримати всі завантажені модулі у двосторонньому списку.
- name - вказує назву модуля. Це ім'я має бути унікальним, оскільки на нього можуть посилатися інші функції ядра, наприклад, щоб вивантажити модуль.

- syms, num_syms та crc використовуються для керування символами, які експортовані модулем.

На рисунку 1.6 зображені вказівники на функції ініціалізації та завершення модулю

```
/* Startup function. */
int (*init)(void);

/* Destruction function. */
void (*exit)(void);
```

Рисунок 1.6 – Функції ініціалізації та завершення модулю

Модулі можуть залежати один в одного. Фунціонал ядра надає спеціальну функцію already_uses, щоб перевірити, чи потрібен модуль A іншому модулю B (рисунок 1.7):

```
/* Does a already use b? */
static int already_uses(struct module *a, struct module *b)
{
        struct module_use *use;

        list_for_each_entry(use, &b->source_list, source_list) {
                if (use->source == a) {
                        pr_debug("%s uses %s!\n", a->name, b->name);
                        return 1;
                }
        }
        pr_debug("%s does not use %s!\n", a->name, b->name);
        return 0;
}
```

Рисунок 1.7 – Функція перевірки залежності модулей

Модулі використовують ELF формат, який містить декілька додаткових розділів, які відсутні в звичайних програмах або бібліотеках. На додаток до кількох розділів, створених компілятором, модулі складаються з наступних розділів ELF формату[22]:

- секції __ksymtab, __ksymtab_gpl та __ksymtab_gpl_future містять таблицю

символів з усіма символами, експортованими модулем.

- __param зберігає інформацію про параметри, прийняті модулем.
- __ex_table використовується для визначення нових записів для таблиці виключень ядра у випадку, якщо код модуля потребує цього механізму.
- Функції ініціалізації та дані зберігаються в .init.text.
- .gnu.linkonce.this_module надає екземпляр структури module, в якому зберігається ім'я модуля та вказівники на функції ініціалізації та звершення.

Деякі з перерахованих вище розділів не можуть бути створені, доки сам модуль і, можливо, всі інші модулі ядра не були скомпільовані, наприклад, розділ, в якому перераховані всі модульні залежності. Оскільки в вихідному коді немає явної інформації про залежність, ядро повинно отримати цю інформацію, аналізуючи невирішені посилання відповідного модуля, а також експортовані символи всіх інших модулів.

Тому для створення модулів виконуються такі шаги:

- По-перше, всі С-файли у вихідному коді модуля складаються в звичайні .o-об'єктні файли.
- Після того, як об'єктні файли були створені для всіх модулів, ядро може їх проаналізувати. Додаткова інформація, яка знайдена під час аналізу зберігається в окремому файлі, який також компілюється у бінарний файл.
- Ці файли пов'язані лінкуються і створюють фінальний модуль.

```
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);

__visible struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
        .name = KBUILD_MODNAME,
        .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
        .exit = cleanup_module,
#endif
        .arch = MODULE_ARCH_INIT,
};

static const char __module_depends[]
__used
__attribute__((section(".modinfo"))) =
"depends=";


MODULE_INFO(srcversion, "E9D346BD6F1E7D4AEF9F5D1");
```

Рисунок 1.8 – vim module.mod.c

Загрузка модулів виконується за допомоги системного виклику init_module (рисунок 1.9).

```
asmlinkage long
sys_init_module(void __user *umod, unsigned long len, const char __user *uargs)
```

Рисунок 1.9 – Системний виклик sys_init_module

Системний виклик init_module приймає три параметри - вказівник на область в адресному просторі користувача, в якому розташований бінарний код модуля (umod), його довжина (len) та вказівник на рядок, який визначає параметри модуля.

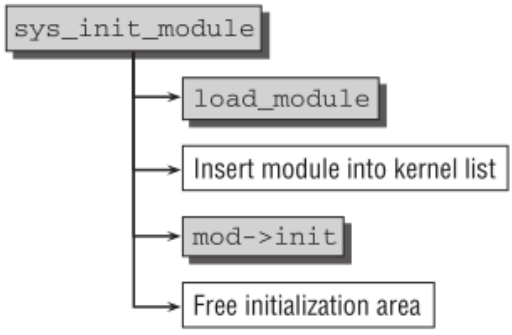

Рисунок 1.10 – Принцип роботи системного виклику init_module[22]

Бінарні дані переносяться в адресний простір ядра за допомогою функції load_module. Аргументи переводяться у формат, який легко аналізувати, і створюється екземпляр структури даних module з усією необхідною інформацією про модуль. Коли екземпляр модуля, створений у функції load_module, був доданий до списку глобальних модулів, ядро ініціалізує модуль та звільняє пам'ять, зайняту даними ініціалізації.

1.3 Принцип роботи load_module

Вся суть завантаження модулю ядра у функції load_module, її головні функції:

- Копіювання даних модуля і його аргументів із простору користувача в тимчасове місце пам'яті в адресний простір ядра; відносні адреси розділів ELF замінюються абсолютними адресами тимчасового зображення.
- Розподіл існуючих розділів на їх остаточні позиції в пам'яті.

```
SYSCALL_DEFINE3(init_module, void __user *, umod,
                unsigned long, len, const char __user *, uargs)
{
        int err;
        struct load_info info = { };

        err = may_init_module();
        if (err)
                return err;

        pr_debug("init_module: umod=%p, len=%lu, uargs=%p\n",
               umod, len, uargs);

        err = copy_module_from_user(umod, len, &info);
        if (err)
                return err;

        return load_module(&info, uargs, 0);
}
```

Рисунок 1.11 – Визначення системного виклику init_module

З рисунку 1.11 можна побачити, що під час виклику init_module, виконується

перевірка, що можливо завантажити модуль при даній конфігурації ядру. Після успішної перевірки виконується копіювання бінарного файлу модулю із простору користувача у простір ядра за допомоги функції copy_module_from_user. Далі викликається функція load_module, її визначення зображене на рисунку 1.12

```
/* Allocate and load the module: note that size of section 0 is always
   zero, and we rely on this for optional sections. */
static int load_module(struct load_info *info, const char __user *uargs,
                       int flags)
{
        struct module *mod;
        long err;
        char *after_dashes;

        err = module_sig_check(info, flags);
        if (err)
                goto free_copy;

        err = elf_header_check(info);
        if (err)
                goto free_copy;

        /* Figure out module layout, and allocate all the memory. */
        mod = layout_and_allocate(info, flags);
        if (IS_ERR(mod)) {
                err = PTR_ERR(mod);
                goto free_copy;
        }

        audit_log_kern_module(mod->name);

        /* Reserve our place in the list. */
        err = add_unformed_module(mod);
        if (err)
                goto free_module;

#ifdef CONFIG_MODULE_SIG
        mod->sig_ok = info->sig_ok;
        if (!mod->sig_ok) {
                pr_notice_once("%s: module verification failed: signature "
                               "and/or required key missing - tainting "
                               "kernel\n", mod->name);
                add_taint_module(mod, TAINT_UNSIGNED_MODULE, LOCKDEP_STILL_OK);
        }
#endif
```

Рисунок 1.12 – Визначення load_module

Під час виконання функції load_module виконуються перевірки підпису

модулю, якщо така конфігурація присутня та перевірка заголовку ELF файлу.

```
/* Sanity checks against invalid binaries, wrong arch, weird elf version. */
static int elf_header_check(struct load_info *info)
{
        if (info->len < sizeof(*(info->hdr)))
                return -ENOEXEC;

        if (memcmp(info->hdr->e_ident, ELFMAG, SELFMAG) != 0
            || info->hdr->e_type != ET_REL
            || !elf_check_arch(info->hdr)
            || info->hdr->e_shentsize != sizeof(Elf_Shdr))
                return -ENOEXEC;

        if (info->hdr->e_shoff >= info->len
            || (info->hdr->e_shnum * sizeof(Elf_Shdr) >
                info->len - info->hdr->e_shoff))
                return -ENOEXEC;

        return 0;
}
```

Рисунок 1.13 – Визначення load_module

Далі адреси всіх розділів у бінарному коді переписуються в абсолютні значення в тимчасовому зображення, проводяться додаткові перевірки та ініціалізації, після цього модуль вже можна використовувати.

```
static int apply_relocations(struct module *mod, const struct load_info *info)
{
        unsigned int i;
        int err = 0;

        /* Now do relocations. */
        for (i = 1; i < info->hdr->e_shnum; i++) {
                unsigned int infosec = info->sechdrs[i].sh_info;

                /* Not a valid relocation section? */
                if (infosec >= info->hdr->e_shnum)
                        continue;

                /* Don't bother with non-allocated sections */
                if (!(info->sechdrs[infosec].sh_flags & SHF_ALLOC))
                        continue;

                /* Livepatch relocation sections are applied by livepatch */
                if (info->sechdrs[i].sh_flags & SHF_RELA_LIVEPATCH)
                        continue;

                if (info->sechdrs[i].sh_type == SHT_REL)
                        err = apply_relocate(info->sechdrs, info->strtab,
                                             info->index.sym, i, mod);
                else if (info->sechdrs[i].sh_type == SHT_RELA)
                        err = apply_relocate_add(info->sechdrs, info->strtab,
                                                 info->index.sym, i, mod);
                if (err < 0)
                        break;
        }
        return err;
}
```

Рисунок 1.14 – Визначення load_module

1.4 MMU

Пам'ять являє собою дуже важливий ресурс, який потребує чіткого управління. Раніше кожна програма, коли виконувала команду лише бачила фізичну пам'ять. Тобто модель пам'яті, що надавалась програмісту, була простою фізичною пам'яттю, із набором адрес від 0 до деякого максимального значення, де кожна адреса відповідав блоку, що містить якусь кількість біт. При таких умовах утримання в пам'яті відразу двох працюючих програм не представлялося можливим, оскільки, якщо одна програма збереже результати своєї операції в комірці 1400, то інша програма могла їх перезаписати.

Для вирішення цієї проблеми було придумано адресний простір, набір адрес, який може бути використаний процесом для звернення до пам'яті. У кожного процесу є свій власний адресний простір, незалежне від того адресного простору, яке належить іншим процесам. Тобто адреса 123 в одній програмі має інше місце у фізичній пам’яті ніж та сама адреса у другій програмі.

На практиці сумарний об’єм фізичної пам’яті може бути менший ніж потрібно для розміщення всіх процесів системи. Для вирішення цієї проблеми існує два способи: свопінг та віртуальна пам'ять. Сутність свопінгу у розміщенні всього процесу у пам'яті, тобто запуск процесу на певний час і зберіганні на диску не потребуючих даних. Непрацюючі процеси будуть зберігатися на диску і займатимуть оперативну пам'ять.

Сутність віртуальної пам'яті у тому, що програма має власний адресний простір, і він розбивається на сторінки пам’яті, які являють собою безперервний діапазон адрес, і відображаються на фізичну пам'ять, і для того запустити програму не потрібно завантажувати всі сторінки пам'яті процесу одночасно. Коли програма посилається на частину свого адресного простору, що знаходиться у фізичній пам'яті, апаратне забезпечення здійснює необхідне відображення дуже швидко. Коли програма посилається на частину свого адресного простору, яке не загружено у фізичну пам'ять, операційна система попереджається про те, що

необхідно загрузити відсутню частину. На пристроях, які не використовують віртуальну пам'ять, віртуальні адреси відображаються безпосередньо на шині пам'яті, що призводить до читання або запису слова фізичної пам'яті з тією ж адресою. При використанні віртуальної пам'яті віртуальні адреси не виставляються безпосередньо на шині пам'яті. Замість цього вони поступають в Memory Management Unit (далі MMU)[8], який відображає віртуальні адреси на адреси фізичної пам'яті, рисунок 1.4.

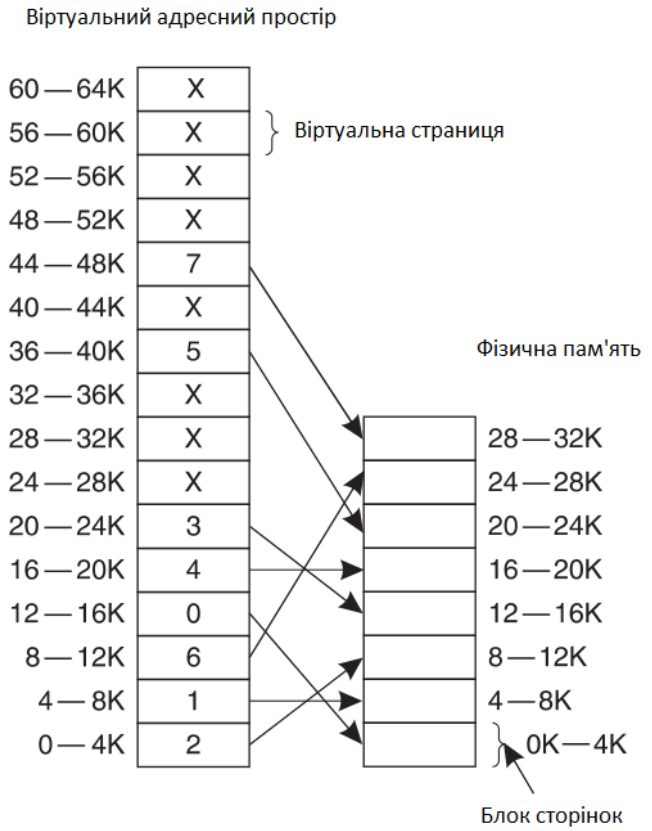

Рисунок 1.15 – Зв'язок між віртуальними адресами і адресами фізичної пам'яті.

Як було сказано раніше, віртуальний адресний простір складається з блоків фіксованого розміру, які називаються сторінками. Відповідні блоки в фізичної пам'яті називаються блоками сторінок. Перенесення інформації між оперативною пам'яттю і диском завжди здійснюється цілими сторінками. Треба зазначити, що система відстежує присутність конкретних сторінок в фізичної пам'яті за рахунок біта присутності-відсутності. Якщо необхідна сторінка не згружена, то диспетчер пам'яті змушує центральний процесор передати керування операційній системі

завдяки системному перериванню page fault. Операційна система вибирає рідко використовуваний сторінковий блок і скидає його вміст на диск. Потім вона завантажує з диска сторінку, на яку було посилання, і поміщає її в щойно звільнився сторінковий блок, вносить зміни в таблиці і заново запускає перервану команду.

Відображення віртуальних адрес на фізичні може бути зведене до наступного: віртуальний адреса ділиться на номер віртуальної сторінки (старші біти) і зміщення (молодші біти). Наприклад, при 16-розрядної адресації і розмірі сторінок 4 Кб старші 4 біти можуть визначати одну з 16 віртуальних сторінок, а молодші 12 біт - зміщення в байтах (від 0 до 4095) всередині обраної сторінки.

Номер віртуальної сторінки використовується в якості індексу всередині таблиці сторінок, який потрібен для пошуку запису для цієї віртуальної сторінки. Із запису в таблиці сторінок береться номер сторінкового блоку. Номер сторінкового блоку приєднується до старших бітів зміщення, замінюючи собою номер віртуальної сторінки, щоб сформувати фізичну адресу, який може бути посланий до пам'яті.

Таким чином, призначення таблиці сторінок полягає в відображенні віртуальних сторінок на сторінкові блоки.

На рисунку 1.5 можна побачити типовий запис у таблицю сторінок. При записі у таблицю ми отримуємо Page frame number. Якщо біт присутності дорівнює 1, то запис дозволений, в іншому випадку ми отримаємо помилку відсутності сторінки. Біт захисту означає тип доступу до цієї сторінки: читання, запис, виконання. Коли в сторінку здійснюється запис, апаратура автоматично встановлює біт модифікації. Цей біт має значення, коли операційна система вирішує регенерувати сторінковий блок. Якщо міститься в ньому сторінка піддавалася модифікації, її потрібно вивантажити на диск. Якщо ж навпаки, від неї можна відмовитися, оскільки її дискова копія не втратила актуальності. Цей біт іноді називається бітом зміни, оскільки він відображає стан сторінки.

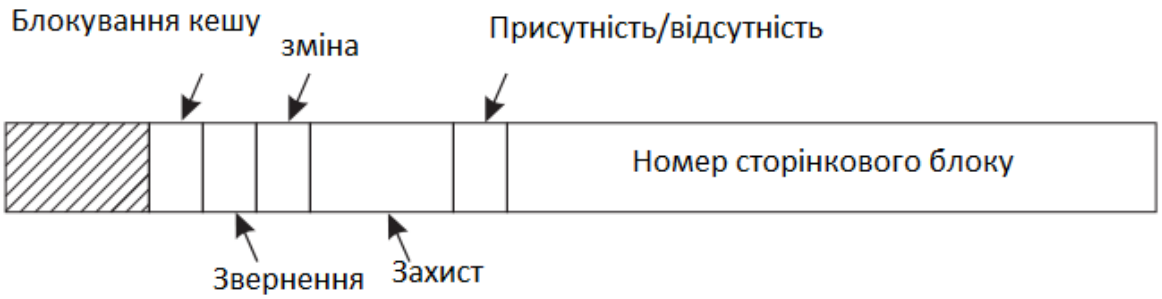

Рисунок 1.16 – Запис у таблицю сторінок.

1.5 Стек виконання

Стек виконання це така структура даних, яка зберігає інформацію про активні підпрограми (subroutine) комп'ютерної програми. Головною його задачею є відслідкування точки повернення з кожної активної підпрограми, тобто адресу повернення. Адресою повернення є наступна після виклику підфункції інструкція в програмному стеку. На рисунках 1.6-1.7 зображено приклад стеку виконання. На строчці 18 асемблерного коду видно, що відбувається виклик функції call() – bl call. Тому адресою повернення буде наступна інструкція, яка зображена на строчці 19, її адреса буде встановлена у регістр LR для ARM платформи, або збережена на стеку для платформи x86.

Стек викликів зазвичай реалізований одним із таких способів:

- В більшій частині платформ стек розташовується в оперативній пам'яті, спеціалізований регістр вказує на його вершину (ESP/RSP для x86/x86-64, SP - ARM);
- Адреса повернення зберігається або в спеціалізованому регістрі (ARM - LR), або в самому стеку (x86).

```c
void call(int a)
{
    (void)a;
}

int main(int argc, char **arv)
{
    int a = 5;
    call(a);
    return 0;
}
```

```asm
call:
  str fp, [sp, #-4]!
  add fp, sp, #0
  sub sp, sp, #12
  str r0, [fp, #-8]
  add sp, fp, #0
  ldmfd sp!, {fp}
  bx lr
main:
  stmfd sp!, {fp, lr}
  add fp, sp, #4
  sub sp, sp, #16
  str r0, [fp, #-16]
  str r1, [fp, #-20]
  mov r3, #5
  str r3, [fp, #-8]
  ldr r0, [fp, #-8]
  bl call
  mov r3, #0
  mov r0, r3
  sub sp, fp, #4
  ldmfd sp!, {fp, pc}
```

Рисунок 1.17 – Приклад стеку виконання.

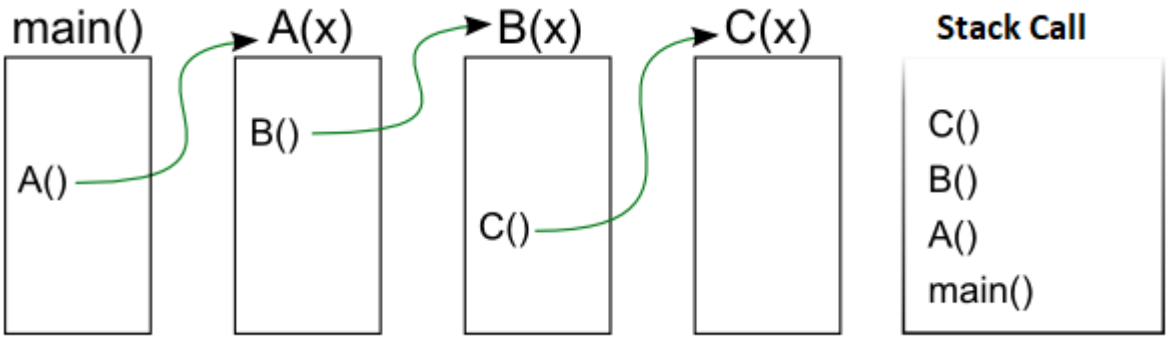

Рисунок 1.18 – Приклад стеку виконання.

При відсутності стека або обмеженості його глибини, вкладені виклики виключені або їх кількість обмежена (рекурсивні виклики). При необхідності більшої вкладеності, стек викликів або його розширення можуть бути реалізовані програмно.

Крім адрес повернення на стеку можуть зберігатися:

- значення регістрів з їх подальшим відновленням (x86)
- аргументи, передані в функцію
- локальні змінні
- інші довільні дані

Змішування таких даних як адреси повернення, збережених вказівників кадрів(ebp – x86, LR - ARM) та звичайних програмних даних - аргументи,

значення до повернення, локальні змінні у стеку викликів може призвести до переповнення буферу на стеку.

1.6 Технології захисту в Linux

В цьому розділі розглянемо ASLR, DEP та W^X, оскільки вони перетинаються з даною роботою.

Завдяки ASLR в адресному просторі процесу випадковим чином змінюється розташування таких важливих структур даних, як образів виконуваного файлу, підвантажуваних бібліотек, купи та стека.

Для використання ASLR виконувані файли потрібно компілювати зі спеціальними параметрами (position-independent executable). В результаті в коді не використовуватимуться постійні адреси, але при цьому:

- збільшиться розмір коду виконуваних файлів;
- збільшиться час завантаження в пам'ять кожного виконуваного файлу;
- виникне додаткова несумісність з ПО і бібліотеками, розробленим під версії ОС без ASLR.

Основний принцип даної технології полягає в усуненні відомих атакуючому адрес адресного простору процесу. Зокрема, адрес, необхідних для того, щоб:

- передати управління на виконуваний код;
- побудувати ланцюжок ROP-гаджетів;
- прочитати (перезаписати) важливі значення в пам'яті.

Для запобігання виконання даних в пам’яті, існує механізм захисту DEP. За допомогою DEP позначає всі осередки пам'яті, які використовуються додатками, як невиконувані, тобто тільки для читання, якщо осередок не містить коду, в явному вигляді. Якщо програма намагається виконати код зі сторінки пам'яті, позначеної як для читання, то, процесор може згенерувати виключення і запобігти виконанню коду.

Таким чином, система завадить шкідливій програмі (наприклад, вірусу) впровадитися в пам'ять комп'ютера та виконувати свій код. DEP дозволяє лише особливим областям пам'яті запускати виконуваний код.

1.7 Особливості ARM архітектури

Далі розглянемо ARM[1] архітектуру і деякі її особливості. ARM або Advanced RISC Machine є представником RISC архітектури. ARM не виробляє свої власні пристрої, а продає ліцензії на розробку процесорів даної архітектури іншим виробникам. Серед основних гравців є Samsung, Apple Qualcomm, NVidia тощо. ARM платформа надає набагато кращу продуктивність, ніж її аналоги (CISC), як з точки зору ефективності, споживання тепла та енергії, особливо на вбудованих пристроях та смартфонах тому вона і широко використовуються в їх розробці.

Мобільні пристрої стали новими улюбленими цілями для зловмисників. Атаки на пристрої можуть включати використання руткітів, троянів та інших зловмисних застосунків, які можуть вкрасти особисту інформацію користувача, який зберігає її в пристрої. Саме для боротьби з цим в ARM існує спеціальне розширення для забезпечення безпеки – TrustZone. ARM-процесори, починаючи з ARM Cortex-A5, підтримують TrustZone.

ARM підтримує сім режимів роботи:

1. Режим користувача (usr): це звичайний стан виконання програми, використовується для виконання більшості прикладних застосунків.

2. Швидкий режим переривання (FIQ): використовується для швидкого переривання.

3. Режим супервізора (SVC): це захищений режим роботи системи.

4. Режим Abort (abt): здійснюється системою пам'яті внаслідок неможливості завантаження будь-якої інструкції або даних.

5. Режим переривання (IRQ): використовується для переривання загального призначення.

6. Режим System (sys): це привілейований режим для операційної системи.

7. Невизначений режим (und): вводиться при виконанні невизначеної інструкції.

З усіх режимів користувацький режим (usr) є єдиним непривілейованим режимом. Привілейовані режими використовуються для обслуговування переривань або виключень, або для доступу до захищених ресурсів.

В ARM існує 16 регістрів доступних регістрів. З R0-R10 – регістри загального використання, а інші – спеціалізовані:

- R11 – frame pointer, зберігає значення поточного стеку;
- R12 – IP регістр, використовується для зберігання тимчасових даних;
- R13 – Stack pointer, зберігає покажчик на вершину стека;
- R14 – Link регістр, зберігає зворотній адрес;
- R15 – Program counter, зберігає адрес інструкції, яка буде виконуватися наступною.

Також не менш важливим є SCTLR (system control register). SCTLR забезпечує контроль над системою, включаючи систему пам'яті. Наприклад, змінюючи значення цього регістру можливо вимкнути MMU або дозволити виконувати зловмисний код, який знаходиться в сторінці пам’яті, яка раніше мала лише право на запис, вимкнувши WXN біт.

Мобільні пристрої стають дедалі більш конфіденційною платформою. На сьогодні мобільні пристрої обробляють широкий спектр конфіденційної інформації, наприклад, біометричні дані або дані платежів та криптографічні ключі тощо. Крім того, сучасні схеми захисту цифрового контенту, наприклад музика або фільми потребують високої ступеня конфіденційності, вимагаючи біль жорсткі гарантій, ніж ті, що пропонуються "звичайною" операційною системою, такою як Android.

Тому ARM розробили таку технологію, що надає так гарантії безпеки і цією технологією є довіреного середовища виконання (англ. Trusted Execution

Environment), воно гарантує безпечне зберігання конфіденційних даних, також безпечну генерацію криптографічних ключів, обчислення криптографічних операцій тощо.

Суть довіреного середовища виконання полягає у поділі системи на два середовища, які не можуть впливати один на одного, а також одне з них використовується в якості довіреного, в ньому зазвичай виконуються безпечні криптографічні обчислення, або зберігаються конфіденційні дані, наприклад, ті самі криптографічні ключі або біометричні дані. Тобто ця технологія дозволяє відокремити систему, в якій виконуються звичайні застосунки користувача, які можуть вплинути на конфіденційні дані, або платіжні застосунки тощо.

Дана технологія нібито поділяє процесор на два середовища — swd (Secure World) та nwd (Normal World). Кожне середовище працює зі власною операційною системою. Зазвичай, для звичайного середовища виробники використовують Linux подібні системи, наприклад Anrdoid або iOS. Для безпечного середовища, зазвичай, використовується легковага збірка Linux-подібного дистрибутива, який містить в собі лише бібліотеки необхідні для роботи в безпечному середовищі.

Реалізація даної технології не потребує ніякого додаткового апаратного забезпечення і підтримується процесорами архітектури ARM, які в даний момент поширені у виробництві мобільних та вбудованих пристроїв.

Перемикання процесора із нормального середовища в безпечне проходить через спеціальний режим процесору — режим монітору. Цей режим доступний тільки із swd і активується при переході з одного середовища в інше.

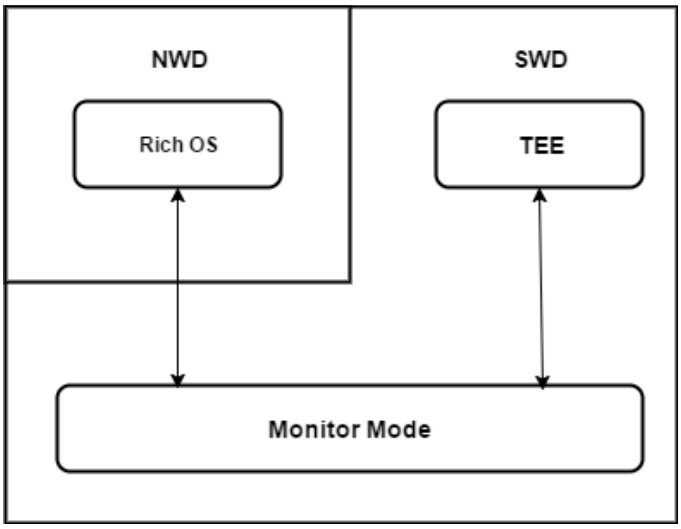

Рисунок 1.19 – Перемикання режиму.

Режим монітору обмежує інші системи в доступі до ресурсів. Наприклад, при переході з безпечного середовища в звичайне, даний режим не дає можливості прочитати звичайному середовищу вміст регістрів із безпечного середовища.

Для того, щоб система могла зрозуміти, в якому режимі працює процесор існує NS-біт, який розташовується в регістрі SCR. Значення NS-біт поширюється по всій системі через AMBA3 AXI шину. Якщо NS-біт встановлений в 1, то процесор працює в звичайному режимі і не має доступу до захищеної пам'яті, а якщо NS-біт встановлений в 0, то процесор працює в безпечному режимі і має доступ до будь-якого обладнання в системі.

Щоб мати змогу перейти в режим монітору, потрібно зробити виклик команди SMC (англ. Secure monitor call), яка може бути виконана тільки в привілейованому режимі, тобто з привілеями ядра, тому, коли процес користувача хоче відправити запрос на зміну середовище, то він повинен спочатку виконати команду SVC (англ. Supervisor call). Виконання цієї команди переводить процесор в привілейований режим і дозволяє виконати команду SMC (Рисунок 1.7):

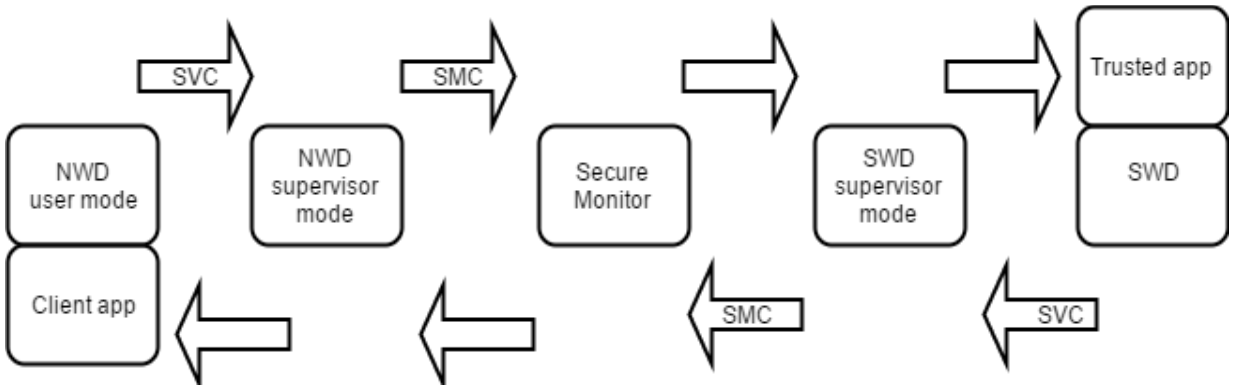

Рисунок 1.20 - Процес перемикання середовища

Також процесор може переходити в безпечний режим при виникненні спеціальних переривань – IRQ та FIQ. FIQ має біль високий пріоритет ніж IRQ. Одночасно може підтримуватися лише одне джерело FIQ. Це допомагає зменшити затримку переривань, оскільки процедура служби переривань може бути виконана безпосередньо без визначення джерела переривання. FIQ та IRQ мають власні збережені регістри, які працюють в режимах FIQ та IRQ, FIQ має більше таких регістрів ніж IRQ.

Але TEE не є панацеєю, вона також має певні вразливі місця[14, 17]. Наприклад, режим монітору (англ. Secure monitor) є найбільш уразливим компонентом TEE. Якщо він буде скомпрометований, то зловмисний код зможе виконуватися в той час, коли процесор виконується в безпечному режимі, а це може дати зловмиснику доступ до конфіденційних даних.

NS-біт. Оскільки TEE не забезпечує апаратний захист для NS-біту, то його можна теж вважати уразливим елементом. Якщо зловмисник зможе скомпрометувати NS-біт, то він буде здатен отримати доступ до безпечної пам’яті та виконувати код в привілейованому режимі. Наприклад, зловмисник може викликати специфічні послідовності переривань IRQ або FIQ, то це може дати зловмиснику доступ до безпечної пам’яті. Однак TEE також може зберігати конфіденційні дані в спеціальних безпечних елементах (англ. Secure element) і зловмисник не буде мати доступу до них. Для того, щоб отримати до них доступ йому потрібно також виконати апаратні атаки на безпечні модулі.

Безпечне сховище (англ. Secure storage). Існує два випадки: генерація даних

в звичайному та безпечному середовищах. У випадку, коли дані можуть генеруватися в безпечному середовищі, а потім зашифрованими зберігатися в звичайному середовищі TEE гарантує, що дані із безпечного середовища виходять в зашифрованій формі, а ключ, яким вони були зашифровані зберігається в безпечній пам’яті. Зловмисник здатен виконати DoS атаку на засоби введення-виведення, щоб заблокувати збереження даних в звичайному середовищі. Дана атака порушує принцип доступності, але конфіденційність та цілісність не компрометуються. В іншому випадку, коли дані генеруються у звичайному середовищі та передаються в безпечне середовище, вони можуть бути скомпрометовані, якщо в системі не передбачено ніяких захисних механізмів. Наприклад, механізм, який шифрує сторінки пам’яті, на яких міститься конфіденційна інформація, перед тим як вони будуть оброблюватися засобами введення-виведення для передачі в безпечне середовище. Приклади компрометації [14, 17].

1.8 ATF, ARM syscall convention,

Trusted Firmware-A (TF-A) забезпечує еталонну реалізацію swd для Armv7-A і Armv8-A, включаючи secure monitor, який виконується у Exception Level 3 (EL3). TF-A реалізує різні інтерфейсні стандарти ARM, але для цієї роботи важливо розглянути цей – SMC calling convention[7].

Існує декілька рівні привілеїв, в яких може знаходитися виконавчий процес:

- EL0 - найнижчий рівень, який використовується для виконання користувацьких застосунків в nwd.
- EL1 - привілейований рівень, який використовується для виконання на рівні ядру операційної системи в nwd.
- EL2 - рівень гіпервізору. Використовується для виконання коду гіпервізора. EL2 завжди знаходиться в незахищеному стані.
- EL3 - рівень, в якому працює secure monitor, який обробляє переходи

між незахищеними та безпечними станами. EL3 завжди знаходиться в безпечному стані.

- S-EL0 - Secure EL0, використовується для виконання довіреного коду програми в безпечному стані.
- S-EL1 - Secure EL1, використовується для виконання коду Trusted OS в безпечному стані.

У архітектурі ARM синхронний контроль передається між нормальним незахищеним станом і безпечним станом за допомогою Secure Monitor Call. SMC виклики генеруються інструкцією SMC і обробляються Secure Monitor.

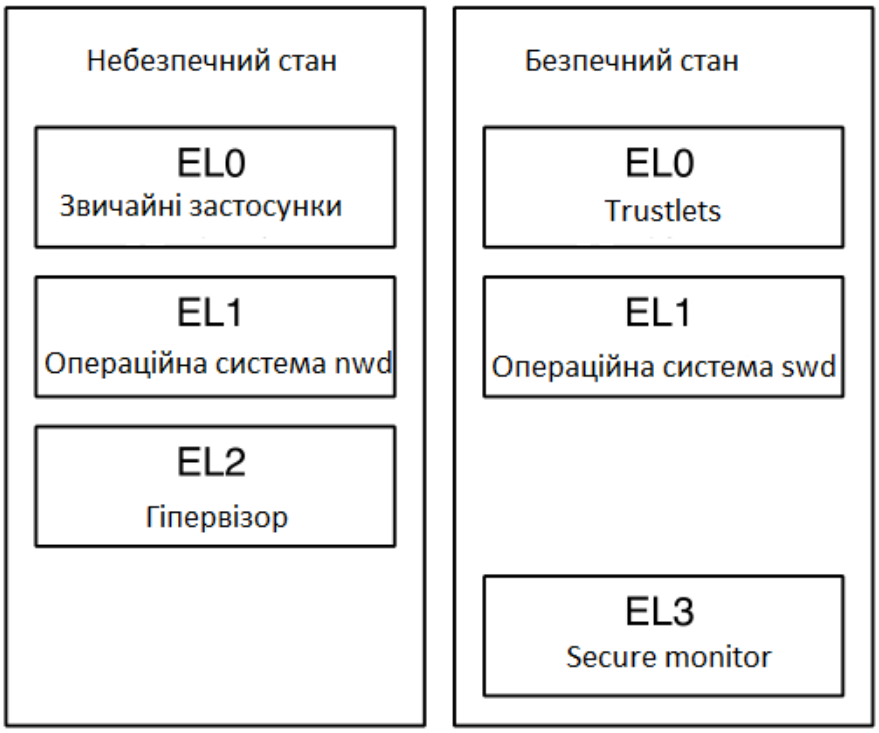

Рисунок 1.21 – Рівні виконання

Дія Secure Monitor визначається параметрами, що передаються через регістри загального застосування. Коли здійснюється виклик SMC (архітектура AArch32), то ідентифікатор функції передається в регістрі R0, аргументи передаються в регістрах R1-R6, результати повертаються в R0-R3, регістри R4-R14 зберігаються за допомогою функції, яка викликається.

1.9 OP-TEE, QEMU

В даній роботі використовувалися OP-TEE[2], та пропрієтарні платформи. OP-TEE - це так звана Trusted Execution Environment, підтримується всіма чипами ARM, які підтримують технологію TrustZone. OP-TEE знаходиться у відкритому доступі. Ця система розроблена відповідно до специфікації Global Platform TEE System Architecture, і відповідно до TEE Internal Core API v1.1.

OP-TEE складається з трьох компонентів:

1. OP-TEE client - який є клієнтським API, що працює в звичайному просторі користувача.
2. OP-TEE driver, який є драйвером ядра Linux, він обробляє зв'язок між nwd у просторі користувача та swd.
3. OP-TEE trusted os, яка є надійною ОС, що працює в swd.

OP-TEE OS складається з двох основних компонентів: самого ядра OP-TEE та колекції бібліотек, призначених для використання в довірених застосунках (Trusted Applications). Ядро OP-TEE виконується на привілейований рівні, довірені застосунки виконують на непривілейованому рівні. Статичні бібліотеки, надані операційною системою OP-TEE, дозволяють довіреним додаткам запускати безпечні служби на більш привілейованому рівні, наприклад сховище даних в довіреному середовищі або криптографічні операції.

Принцип роботи додатку, який використовує функціонал довіреного середовища полягає в наступному - звичайний застосунок, який реалізовано для використання в нормальному середовищі використовує клієнтську API TEE, яка дозволяє взаємодіяти звичайному застосунку з trustlet (Рисунок 3.4):

Trustlet працює в безпечному середовищі, та надає такий функціонал звичайному застосунку:

- шифрування / розшифрування даних
- зберігання даних у безпечному сховищі (англ. Secure storage)

Довірені програми динамічно завантажуються ядром OP-TEE в безпечне

середовище, коли звичайний застосунок із нормального середовища захоче взаємодіяти з одним із них. Це схоже на те, як ядро Linux може динамічно завантажувати модулі ядра, хоча на відміну від Linux, в OP-TEE trustlets працюють на більш низькому привілейованому рівні процесора, ніж основний код OP-TEE.

Кожен trustlet підписується 2048-бітним RSA ключем. На даному етапі технологія OP-TEE підтримує лише підпис всіх trustlets одним ключем. Ядро OP-TEE також підписане даним ключем. В даний момент розроблюється інший спосіб підписування trustlets, але в даному випадку, якщо зловмисник зможе отримати ключ, яким підписано trustlet, то він може створити свій trustlet і підписати його. Завдяки тому, що trustlets підписуються вони можуть бути збережені в звичайному середовищі і перед кожною загрузкою в безпечне середовище вони перевіряються.

Для тестування використовувався емулятор QEMU [3]. Даний емулятор дозволяє розробляти і тестувати програми для ARM архітектури процесора.

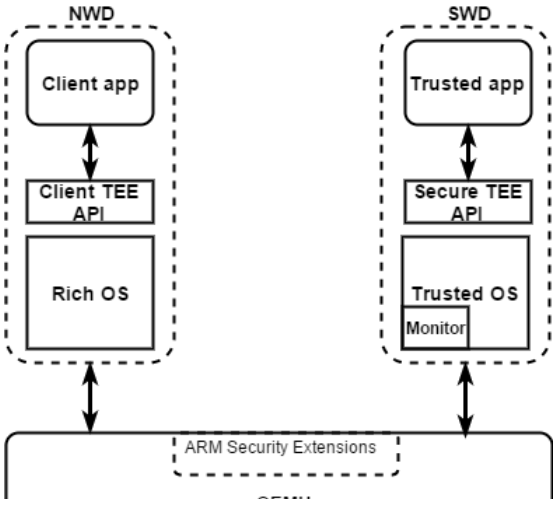

Рисунок 1.22 - Емуляція TEE

Висновки до розділу 1

В даному розділі були розглянуті основні принципи будови операційної системи Linux, основні її функції. Було визначено, що операційна система є програмою, яка управляє всіма іншими програмами, якими користується звичайний користувач, представляє йому графічний інтерфейс. Зі сторони розробника, операційна система це певний інтерфейс, який дозволяє користуватися всіма ресурсами системи. Основними функціями операційної системи є:

- Багатозадачність. У багатозадачній операційній системі, в якій одночасно може працювати декілька програм, операційна система визначає, які програми повинні працювати в якому порядку та скільки часу потрібно допускати для кожної програми.
- Контроль пам’яті. Операційна система керує розподілом внутрішньої пам'яті між процесами.
- I\O. Обробка вхідних та вихідних даних. Введення/Виведення на приєднані апаратні пристрої, такі як жорсткі диски, принтери та комутовані порти тощо.
- Взаємодія між процесами.
- та інші.

Ядро Linux є важливою частиною Linux-подібної операційної системи. Воно підтримує динамічне завантаження модулів ядра, які можуть розширювати доступний фунціонал ядра Linux. Під час завантаження модулю, проходить перевірка його сумісності та його підпису, далі йде його завантаження та ініціалізація. За це відповідає системний виклик init_module.

В операційній системі існують механізми захисту пам’яті. Вони ускладнюють експлуатацію вразливостей (buffer overflow, integer overflow тощо).

Одними із розглянутих технік є ASLR та DEP. Принцип роботи ASLR полягає в тому, що він змінює початкові адреси стеку, купи, бібліотек, тому зловмиснику буде важче написати експлоіт. DEP позначає всі сторінки пам’яті, в

яких немає коду прапорцем READ. Це не дає змоги виконувати вразливий код зі стеку, який туди помістили, наприклад, після експлуатації вразливості buffer overflow.

Також було розглянуто особливості архітектури ARM. Були розглянуті основні доступні користувачу регістри та приклади їх використання. Було розглянуто технологію Trusted Execution Environment. Основна її суть полягає в тому, що вона поділяє роботу системи на два світи: безпечний та небезпечний. Перемикання між ними виконується завдяки secure monitor, а саме завдяки виклику команди SMC. Однією з її переваг даної технології те, що безпечний світ працює в на іншому рівні привілеїв. І звичайні застосунка та навіть ядро операційної системи не зможуть впливати на безпечне середовище без певних налаштувань.

2 ОГЛЯД ЗАГРОЗ ТА АНАЛІЗ ІСНУЮЧИХ МЕХАНІЗМІВ ЗАХИСТУ

2.1 Типові вразливості та методи захисту в user mode

Для мобільних пристроїв так само, як і для стаціонарних комп'ютерів характерні звичайні загрози інформаційній безпеці, пов'язані з вразливостями у програмному забезпеченні, яке, власне, використовує користувач. Зловмисник може виявити програмне забезпечення своєї жертви, надіслати їй спеціально сформовані дані, які використають вразливість в цій системі і зламають її, а після цього зловмисник зможе роботи все, що захоче. Наприклад, застосунок може мати вразливість переповнення масиву (рисунок 2.1).

```c
#include <stdio.h>
#include <stdlib.h>

void never_execute()
{
    printf("It should never be executed");
    return;
}

void vulnerable(char *in)
{
    char buf[10];
    strcpy(buf, in);
}

int main(int argc, char *argv[])
{
    vulnerable(argv[1]);
    return 0;
}
```

Рисунок 2.1 – Приклад вразливості переповнення масиву.

Це є простим прикладом для розуміння проблеми, даний приклад розглядається для платформи ARM. Отже, на рисунку 2.1 можна побачити 3 функції: main, vulnerable, never_execute. Функція main є глобальною функцією і є точкою старту програми. Її аргументами є змінна argc та argv. Argc - невід'ємне число, означає кількість аргументів, переданих програмі з оточення, в якому

запустили програму. Argv - аргументи, передані програмі з оточення, в якому запустили програму. Функція main викликається при старті програми після ініціалізації нелокальних об'єктів зі статичної тривалістю зберігання.

Також ми бачимо тіло функції never_execute, яка лише виводить строку “it should never be executed”, але ця функція ніде не використовується, вона лише об’явлена.

В тілі функції main ми бачимо виклик функції vulnerable, у функцію vulnerable передається argv, що вже не є безпечним. Для порозуміння чому саме не безпечне, потрібно розглянути сегменти пам’яті процесу.

Розглянемо звичайний процес в операційній системі, наприклад застосунок з рисунку 2.1. В ньому ми маємо 3 функції, а в одній із них локальну змінну. Це лише приклад, в реальних процесах використовується набагато більше змінних, функцій тощо. Як видно із рисунку 2.1 дані процесу (функції, змінні) поділені на сегменти:

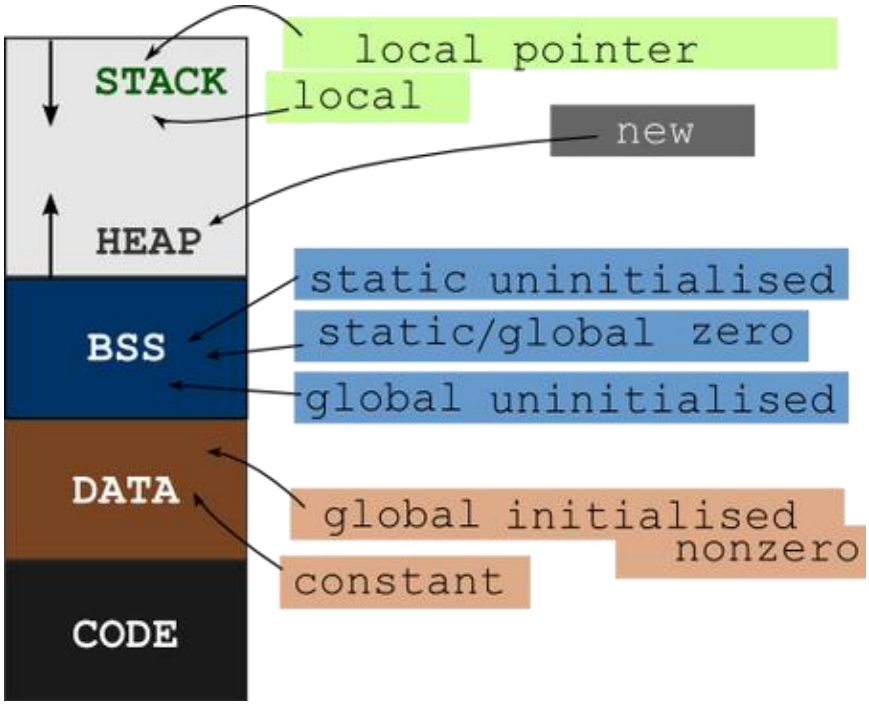

Рисунок 2.2 – сегменти пам’яті

- Stack – сегмент даних, який містить локальні змінні, адреси повернення, stack canary тощо. Об’єкти , які зберігаються на стеку мають певний час життя, після виходи процесу виконання із функції, все змінні об’явлені на стеку перестають існувати (вони існують в пам’яті, але їх використання іншою функцією може призвести до segmentation fault або undefined behavior).
- Heap – вся динамічно виділена пам’ять зберігається в даному сегменті. Приклад функцій, які працюють з динамічною пам’яттю: new, malloc, calloc, realloc, та free.
- Text або Code segment – містить коду програми.
- Data –містить будь-які ініціалізовані глобальні або статичні змінні.
- BSS –містить не ініціалізовані дані.

В даному випадку для нас важливий стек, отже бачимо, що функція vulnerable має локальну змінну buf розміром 10 байтів і виділену на стеку, також після об’явлення цієї змінної, виконується виклик функції strcpy (рисунок 2.3).

```
void vulnerable(char *in)
{
    char buf[10];
    strcpy(buf, in);
}
```

```
vulnerable:
  stmfd sp!, {fp, lr}
  add fp, sp, #4
  sub sp, sp, #24
  str r0, [fp, #-24]
  sub r3, fp, #16
  mov r0, r3
  ldr r1, [fp, #-24]
  bl strcpy
  sub sp, fp, #4
  ldmfd sp!, {fp, pc}
```

Рисунок 2.3 – Функція vulnerable (ARM gcc-4.5.4)

Функція strcpy() копіює дані із другого аргументу в перший, також треба зазначити, що ця функція копіює строки, і довжина другого аргументи визначається підрахунком символів до нульового байту цієї строки. Ця функція не

безпечна, оскільки в ній неможливо задати розмір вхідних даних для копіювання, і зловмисник може подати строку на вхід цієї функцію, яка більше за розміром буферу, в який вона буде копіюватися. Ця вразливість називається buffer overflow.

У переповненні буфера метою зловмисника є зміна адресу повернення, і контролювання поточного потоку програми за допомогою керування регістром LR(у випадку ARM платформи). Як тільки зловмисник опанував зворотним адресом, він можемо контролювати всю програму і змусити її виконувати бажані йому дії.

На рисунку 2.4 зображено приклад функції main у асемблерному коді. В тілі функції main відбувається виклик функції vulnerable – bl vulnerable, а для того, щоб процес виконання повернувся назад у функцію main, у регістр LR встановлюється адрес наступної після виклику bl vulnerable інструкції (адрес повернення).

Якщо зловмисник подасть на вхід до функції vulnerable строку довжиною більше 10 байтів, то він здатен переписати адрес повернення і замість нього встановити інший, наприклад адрес першої інструкції функції never_execute. Розглянемо приклад експлуатації.

```
main:
  stmfd sp!, {fp, lr}
  add fp, sp, #4
  sub sp, sp, #8
  str r0, [fp, #-8]
  str r1, [fp, #-12]
  ldr r3, [fp, #-12]
  add r3, r3, #4
  ldr r3, [r3, #0]
  mov r0, r3
  bl vulnerable
  mov r3, #0
  mov r0, r3
  sub sp, fp, #4
  ldmfd sp!, {fp, pc}
```

Рисунок 2.4 – Main у асемблерному коді (ARM gcc-4.5.4)

Запустимо приклад у gdb на емуляторі QEMU, та переглянемо асемблерний код функцій main, vulnerable, never_execute (рисунки 2.5 – 2.7):

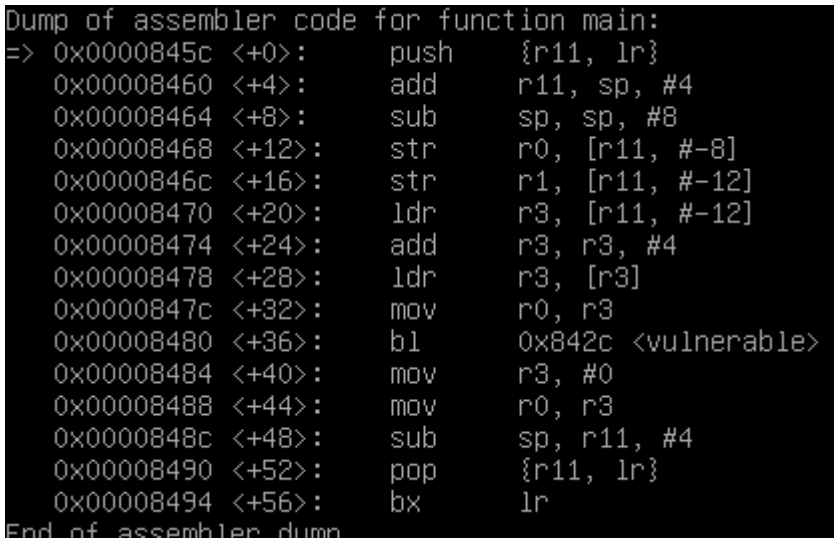

Рисунок 2.5 – Дамп функції main

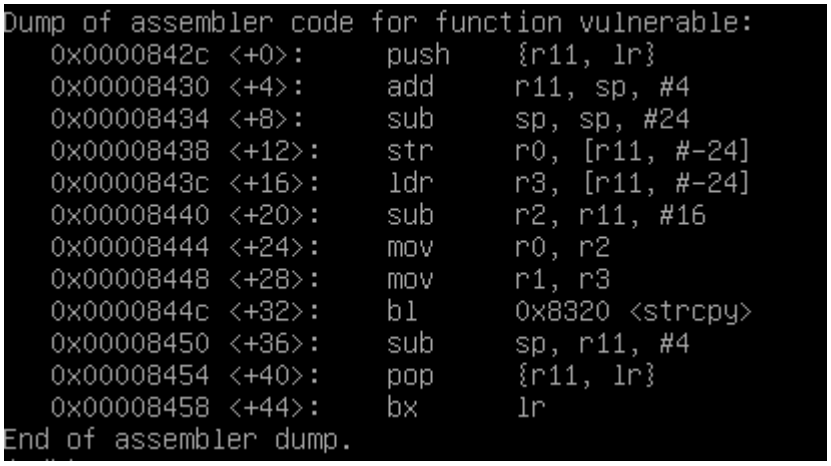

Рисунок 2.6 – Дамп функції vulnerable

На рисунку 2.5 можна побачити виклик функції vulnerable:

0x00008480 <+36>: bl 0x842c <vulnerable>

І як було наведено раніше, щоб повернути виконання у викликавшу функцію, тобто у main, у регістр LR встановлюється значення наступної інструкції – 0x00008484.

```
Dump of assembler code for function never_execute:
   0x00008408 <+0>:     push    {r11, lr}
   0x0000840c <+4>:     add     r11, sp, #4
   0x00008410 <+8>:     ldr     r3, [pc, #16]   ; 0x8428 <never_execute+32>
   0x00008414 <+12>:    mov     r0, r3
   0x00008418 <+16>:    bl      0x8314 <printf>
   0x0000841c <+20>:    sub     sp, r11, #4
   0x00008420 <+24>:    pop     {r11, lr}
   0x00008424 <+28>:    bx      lr
   0x00008428 <+32>:    andeq   r8, r0, r12, lsl r5
End of assembler dump.
```

Рисунок 2.7 – Дамп функції never_execute

На рисунку 2.8 можно побачити, що PC показує на інструкцію 0x00008444 <+24> : mov r0, r2, а за нею йде інструкція mov r1, r3. Ці команди встановлюють певні значення у регістри r0, r1, дані регістри будуть використовуватися у якості аргументів до функції, яка далі буде викликатися, тобто strcpy.

```
Dump of assembler code for function vulnerable:
   0x0000842c <+0>:     push    {r11, lr}
   0x00008430 <+4>:     add     r11, sp, #4
   0x00008434 <+8>:     sub     sp, sp, #24
   0x00008438 <+12>:    str     r0, [r11, #-24]
   0x0000843c <+16>:    ldr     r3, [r11, #-24]
   0x00008440 <+20>:    sub     r2, r11, #16
=> 0x00008444 <+24>:    mov     r0, r2
   0x00008448 <+28>:    mov     r1, r3
   0x0000844c <+32>:    bl      0x8320 <strcpy>
   0x00008450 <+36>:    sub     sp, r11, #4
   0x00008454 <+40>:    pop     {r11, lr}
   0x00008458 <+44>:    bx      lr
End of assembler dump.
(gdb) i r $r2
r2             0xbefffc94       3204447380
(gdb) x/10x 0xbefffc94
0xbefffc94:     0x000084ec      0x00000000      0x0000849c      0xbefffcb4
0xbefffca4:     0x00008484      0xbefffe04      0x00000002      0x00000000
0xbefffcb4:     0xb6eb3694      0xb6fd5000
(gdb) i r $r3
r3             0xbefffefd       3204447997
(gdb) x/10x 0xbefffefd
0xbefffefd:     0x41414141      0x42424242      0x54004343      0x3d4d5245
0xbeffff0d:     0x756e696c      0x48530078      0x3d4c4c45      0x6e69622f
0xbeffff1d:     0x7361622f      0x55480068
```

Рисунок 2.8 – Дамп функції vulnerable та регістрів r2, r3

До регістру r0 поміщується адреса пам’яті, в яку буде скопійована

переданий користувачем рядок (із r2), тобто це адреса змінної buf, вона виділена на стеку. Командою x/10x 0xbefffc94 можна побачити виділене місце під змінну. Рядок переданий користувачем зберігається на стеку і його можна побачити за допомогою команди x/10x 0xbefffefd або x/10x $sp – це значення 0x41414141, 0x42424242, 0x4343.

За допомогою команди x 0xbefffca4 можна також побачити адресу повернення до функції main – 0x00008484. І щоб її переписати, нам потрібно подати на вхід рядок довжиною 20 байтів.

Після загрузки регістрів r0, r1 викликається функція strcpy:

0x0000844c <+32>: bl 0x8320 <strcpy>.

Перехід у дану функцію ми бачимо на рисунку 2.9.

```
Dump of assembler code for function strcpy:
=> 0x00008320 <+0>:     add     r12, pc, #0, 12
   0x00008324 <+4>:     add     r12, r12, #8, 20
   0x00008328 <+8>:     ldr     pc, [r12, #808]!
End of assembler dump.
(gdb) si
0x00008324 in strcpy ()
(gdb)
0x00008328 in strcpy ()
(gdb)
0x00008300 in ?? ()
(gdb) disassemble
No function contains program counter for selected frame.
(gdb)
No function contains program counter for selected frame.
(gdb) si
0x00008304 in ?? ()
(gdb)
0x00008308 in ?? ()
(gdb)
0x0000830c in ?? ()
(gdb)
0xb6fee1a4 in ?? () from /lib/ld-linux.so.3
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x45454544 in ?? ()
```

Рисунок 2.9 – Перетирання адреси повернення

Після її завершення виконання повертається до функції vulnerable, яка завершує свою роботу виконанням команди bx lr. Так як ми перезаписали адресу

повернення на наше значення, то bx lr переходить до невідомої адреси 0x45454544 (рисунок 2.9). Якщо ми замінено останні 4 байти нашого рядка на адресу першої інструкції функції never_execute, то виконання перейде до неї (рисунок 2.10).

Рисунок 2.10 – Виконання функції never_execute

Це лише простий приклад експлуатації, замість адреси функції never_execute можливо передати адресу, за якою лежить зловмисний payload, виконання якого може надати доступ до root прав.

Один із методів захисту від даної вразливості є компіляція програми зі спеціальним параметром(compilation flag) - stack_protector. В залежності від компілятора можуть бути різні комбінації даних параметрів. Ці параметри компіляції унеможливлюють таку просту експлуатацію даної вразливості, також існують stack canaries, ASLR, W^X тощо.

Найчастіше зловмисники зустрічаються із таким типом захисту як W^X, тобто не можливо вже перевести виконання на свій payload, який зберігається на

стеку тому, що тепер заборонено виконувати код у пам’яті, яка має привілеї на запис.

Але і з цим можна боротися. Існує така техніка атак як ret2libc або ROP. Її суть полягає в тому, що тепер потрібно знайти в бібліотеках, які злінковані із вразливою програмою шматки коду, який має привілеї на виконання. Такі шматки коду називаються ROP-гаджетами. На ARM платформі не можлива атака ret2libc[11], оскільки в ARM інша конвенція виклику функцію – аргументи потрібно передавати через регістри на відміну від платформи x86, де їх можна передавати через стек і, власне, на x86 ця атака можлива. В ARM використовуються специфічна техніка ROP. Потрібно знайти такі гаджети, які зможуть встановити задані користувачем значення в певні регістри. Корисними гаджетами є ті, в яких є такі інструкції:

- LDM (наприклад загрузити певне значення із пам’яті в регістр);
- ADD;
- POP (наприклад взяти значення зі стеку і помістити його в регістр LR);

На рисунку 2.11 приклад корисного гаджету. PC+32 – PC+44:

```
ldm sp, {r0, r1}
add sp, sp #12
pop {lr}
bx lr
```

```
Dump of assembler code for function erand48:
   0xb6ecf128 <+0>:     push    {lr}            ; (str lr, [sp, #-4]!)
   0xb6ecf12c <+4>:     ldr     r3, [pc, #36]   ; 0xb6ecf158 <erand48+48>
   0xb6ecf130 <+8>:     ldr     r1, [pc, #36]   ; 0xb6ecf15c <erand48+52>
   0xb6ecf134 <+12>:    sub     sp, sp, #12
   0xb6ecf138 <+16>:    add     r3, pc, r3
   0xb6ecf13c <+20>:    add     r1, r3, r1
   0xb6ecf140 <+24>:    mov     r2, sp
   0xb6ecf144 <+28>:    bl      0xb6ecf2e0 <erand48_r>
   0xb6ecf148 <+32>:    ldm     sp, {r0, r1}
   0xb6ecf14c <+36>:    add     sp, sp, #12
   0xb6ecf150 <+40>:    pop     {lr}            ; (ldr lr, [sp], #4)
   0xb6ecf154 <+44>:    bx      lr
   0xb6ecf158 <+48>:    andseq  r5, r0, r0, asr #29
   0xb6ecf15c <+52>:    andeq   r3, r0, r0, lsr r2
End of assembler dump.
```

Рисунок 2.11 – Виконання функції never_execute

Тут ми одразу можемо оволодіти регістрами r0, r1, lr. Потрібно лише покласти необхідні значення на стек у певному порядку. Щоб наш експлоіт запрацював потрібно знайти необхідні гаджети, як на рисунку 2.11, а потім побудувати ланцюг із цих гаджетів. Але ми знов наткнемось на системи захисту, які нам буду заважати використовувати ці гаджети – це stack canaries та ASLR.

Stack canaries являють собою випадкове значення, яке буде розміщене безпосередньо перед адресою повернення на стеку і, якщо зловмисник змінить це значення – програма зупине виконання із помилкою – Segmentation fault.

Тобто в нашому прикладі буде не можливо переписати адресу повернення, оскільки потрібно ще визначити або вгадати 4 байти перед цією адресою, що досить ускладнює задачу експлуатації. Крім цього, ці випадкові значення змінюються при кожному запуску програми, завдяки ASLR. Тобто не достатньо один раз їх зчитати з пам’яті.

ASLR – випадково змінює адресний простір процесу, тобто випадковим чином змінює адреси сегментів пам’яті: стек, купа. А також адреси бібліотек.

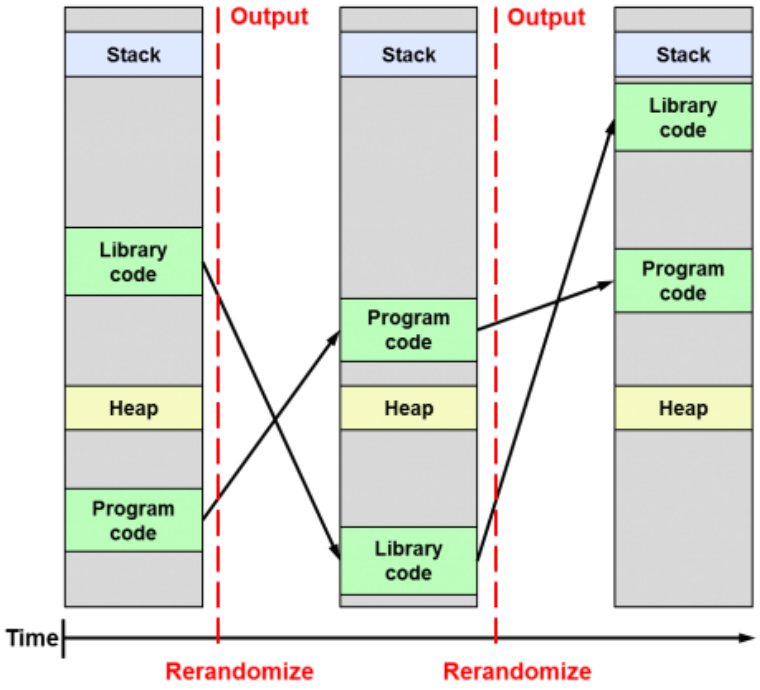

Рисунок 2.12 – Принцип роботи ASLR

Це ще одна перешкода, оскільки знайшовши адреси потрібних нам гаджетів

у певній бібліотеці, ми не зможе їх використати, оскільки за кожним запуском застосунку ці значення змінюються.

Але із цим можна впоратися. Не всі бібліотеки компілюються із використанням ASLR, оскільки його використання впливає на продуктивність, а також існують вразливості і в самому механізмі ASLR[11, 13]. Також, якщо пощастить, то в вразливій програмі можуть бути витоки пам'яті, завдяки яким можна визначити реальні адреса гаджетів. Найпростішим прикладом такої вразливості є format string vulnerability.

Розглянемо приклад такої вразливості (рисунок 2.13). Format string вразливість дозволяє зловмиснику прочитати увесь стек, і таким чином визначити потрібні йому адреси, а також stack canaries.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    printf(argv[1]);
    return 0;
}
```

Рисунок 2.13 – Приклад format string вразливості

На рисунку 2.13 ми бачимо виклик функції printf без вказання формату виводу. Строки форматування - це рядки ASCIIZ, які використовуються для вказівки, як потрібно представити дані. Наприклад, printf("%d", i) - %d вказує, що потрібно представити і у вигляді цілого числа. Таким чином, ця функція використовує рядок формату, щоб перетворити типи даних в рядок, який ми побачимо на екрані. Стрічка формату вводу визначає кількість аргументів, які слід зчитувати зі стеку. Отже, у нашому випадку, якщо ми передаємо наш рядок в якості рядку формату, то функція printf може бути обдурена і буде зчитувати значення зі стеку.

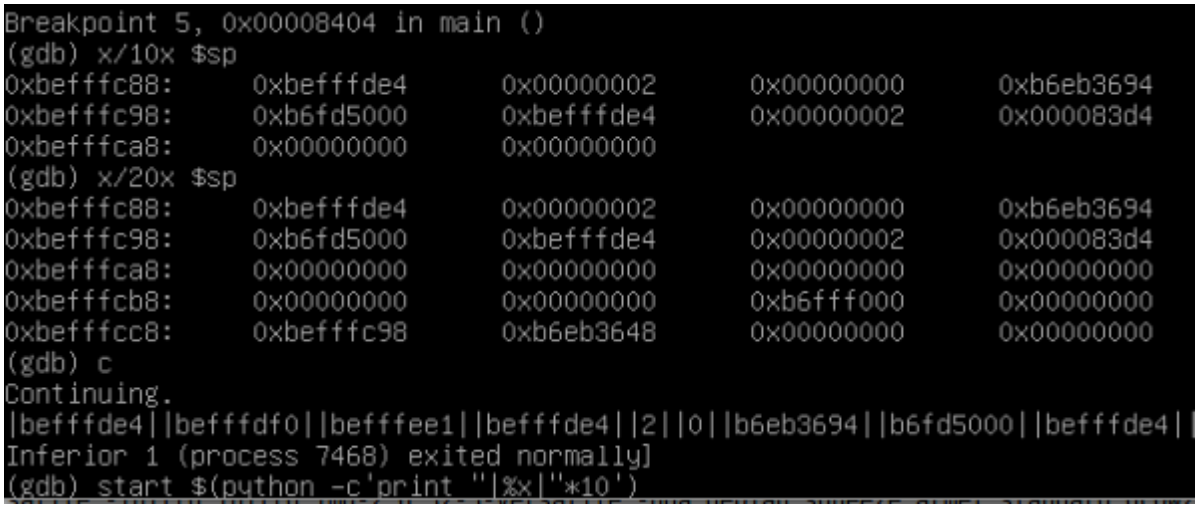

Рисунок 2.14 – Зчитування стеку

Таким чином, подавши на вхід формат строку- ”%x”, ми можемо прочитати стек.

2.2 Типові вразливості в privileged mode

Під вразливостями в privileged mode маються на увазі вразливості в ядрі Linux. У цій частині ми розглянемо декілька уразливості в драйверах Linux. Під час реалізації драйверів ядра Linux розробник може зареєструвати файл драйвера пристрою, який як правило, буде зареєстрований у каталозі /dev/. Цей файл може підтримувати всі звичайні функції: open, read, write, mmap, close тощо. Операції, які підтримуються драйвером описані в структурі file_operations, яка містить функцій вказівники, по одному для кожної операції, і розробник може написати свою функцію, а потім зареєструвати її в цій структурі. Як показано на рисунку 2.15, існує велика кількість операцій, яку розробник може реалізувати, та кожна з них може містити вразливості, які можуть призвести до серйозних проблем в безпеці системи. Загалом вразливості такі ж самі, як і в звичайних програмах, але їх експлуатація може надати зловмиснику права root. Наприклад, розглянемо власну реалізацію функції mmap. Варто зазначити, що досить багато розробників драйверів Linux реалізовують власну функцію mmap[16].

Функція mmap відображає length байтів, починаючи з зміщення offset файлу,

визначеного файловим дескриптором fd, в пам'ять, починаючи з адреси start. Останній параметр (адреса) необов'язковий, і зазвичай дорівнює 0. Справжнє місце розташування відображених даних повертається функцією mmap, і ніколи не буває рівним 0. Після реалізації функції, розробник її реєструє.

```
struct file_operations {
struct module *owner;
loff_t(*llseek) (struct file *, loff_t, int);
ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
ssize_t(*read_iter) (struct kiocb *, struct iov_iter *);
ssize_t(*write_iter) (struct kiocb *, struct iov_iter *);
int(*iterate) (struct file *, struct dir_context *);
int(*iterate_shared) (struct file *, struct dir_context *);
unsigned int(*poll) (struct file *, struct poll_table_struct *);
long(*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long(*compat_ioctl) (struct file *, unsigned int, unsigned long);
int(*mmap) (struct file *, struct vm_area_struct *);
int(*open) (struct inode *, struct file *);
int(*flush) (struct file *, fl_owner_t id);
int(*release) (struct inode *, struct file *);
int(*fsync) (struct file *, loff_t, loff_t, int datasync);
int(*fasync) (int, struct file *, int);
int(*lock) (struct file *, int, struct file_lock *);
ssize_t(*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
int);
unsigned long(*get_unmapped_area)(struct file *, unsigned long, unsigned
long, unsigned long, unsigned long);
int(*check_flags)(int);
int(*flock) (struct file *, int, struct file_lock *);
ssize_t(*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
ssize_t(*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
int(*setlease)(struct file *, long, struct file_lock **, void **);
long(*fallocate)(struct file *file, int mode, loff_t offset,loff_t len);
void(*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
unsigned(*mmap_capabilities)(struct file *);
#endif
ssize_t(*copy_file_range)(struct file *, loff_t, struct file *, loff_t,
size_t, unsigned int);
int(*clone_file_range)(struct file *, loff_t, struct file *, loff_t,u64);
ssize_t(*dedupe_file_range)(struct file *, u64, u64, struct file *, u64);
};
```

Рисунок 2.15 – структура file_operations (Linux Kernel 4.9)

На рисунку 2.16 наведено приклад реєстрації власних функцій в структурі file_operations.

```
static const struct file_operations my_operations = {
.open = my_open,
.read = my_read,
.llseek = my_lseek,
.release = my_release,
.mmap = my_mmap
};
```

Рисунок 2.16 – реєстрація callback

Основною метою обробника mmap полягає в тому, щоб прискорити обмін даними між програмами із простору користувача і простором ядра. Ядро може ділитися деяким фізичним діапазоном пам'яті безпосередньо з адресним простором користувача. Процес користувача може змінити цю пам'ять безпосередньо, без необхідності виклику інших системних викликів.

Приклад реалізації обробника mmap (рисунок 2.17):

```
static int my_open(struct inode *inodep, struct file *fptr)
{
    printk("Device has been opened\n");
    fptr->private_data = kzalloc(0x10000, GFP_KERNEL);
    if (fptr->private_data == NULL)
        return -1;
    return 0;
}

static int my_mmap(struct file *fptr, struct vm_area_struct *vma)
{
    printk("My mmap\n");
    if (remap_pfn_range(vma, vma->vm_start, virt_to_pfn(fptr->private_data),
        vma->vm_end - vma->vm_start, vma->vm_page_prot))
    {
        printk("Failed to mmap \n");
        return -EAGAIN;
    }
    return 0;
}

static struct file_operations my_ops =
{
    .open = my_open,
    .mmap = my_mmap,
};
```

Рисунок 2.17 – Приклад реалізації mmap

Під час відкриття драйверу, приклад якого наведено вище, буде викликана функція my_open, яка просто виділить буфер розміром 0x10000 байт і збереже вказівник на нього у полі fptr->private_data. Після цього, якщо буде виконано виклик функції mmap, то функція буде використовуватися функція my_mmap. Ця функція просто визиває функцію remap_pfn_range, яка створить нове відображення в адресному просторі процесу, який зв'яже буфер fptr->private_data з vma->vm_start, розміром якого визначений як vma->vm_end - vma-> vm_start.

Розглянемо уразливості в цьому коді. Представлений вище код (рисунок

2.17) являє собою загальний підхід до реалізації обробника mmap. Головною проблемою в цьому коді є відсутність перевірки значення vma->vm_end - vma->vm_start, результат цієї операції безпосередньо передається до remap_pfn_range як параметр розміру. Це означає, що зловмисний процес може встановити будь-який розмір для відображення. В цьому випадку це дозволить процесу з користувацького простору, відобразити всю фізичну пам'ять розташовану після буфера fptr->private_data, тобто всю пам'ять ядра. Це означає, що зловмисний процес зможе читати/писати в пам'ять ядра із користувацького простору.

Інший приклад вразливості в реалізації mmap зображено на рисунку 2.18.

```
if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
    vma->vm_end - vma->vm_start, vma->vm_page_prot))
{
    printk("Failed to mmap \n");
    return -EAGAIN;
}
```

Рисунок 2.18 – Приклад вразливої реалізації mmap

З рисунку 2.17 ми знаємо, що аргумент функції vma контролюється користувачем функції mmap, а в прикладі, на рисунку 2.18 значення vma-> vm_pgoff безпосередньо передається функції remap_pfn_range. Це призведе до того, що зловмисний процес може передавати довільну значення (фізичну адресу) до функції mmap, що дозволить отримати доступ до всієї пам'яті ядра з простору користувача. Всі значення, які контролюються користувачем повинні перевірятися.

Також доволі часто в коді можна зустріти складні розрахунки та перевірки суми розмірів вхідних аргументів, допустимих значень тощо. Їх доволі складно розуміти, а також в них може бути помилка, що може призвести до зламу системи, наприклад integer overflow. В наступному прикладі наведено код із цією вразливістю (рисунок 2.19).

```
static int mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned int vma_size = vma->vm_end - vma->vm_start;
    unsigned int offset = vma->vm_pgoff << PAGE_SHIFT;

    if (vma_size + offset > 0x10000)
    {
        printk("Too large a chunk of memory\n");
        return -EAGAIN;
    }
```

Рисунок 2.19 – Integer overflow

На рисунку 2.19 наведений код із вразливістю integer overflow. Переповнення відбувається, коли зловмисний процес запускає системний виклик mmap із розміром (на рисунку 2.19 це vma_size), рівним 0xfffa000 та зсувом (на рисунку 2.19 - offset) 0xf0006. Переповнення відбудеться, оскільки offset буде перетворено у 0xf0006000, а сума 0xfffa000 і 0xf0006000 дорівнює 0x100000000. Оскільки максимальне значення беззнакового цілого числа - 0xffffffff, тому найбільш значний біт буде дорівнювати 0, і остаточне значення суми буде всього 0x0. Тому перевірка вхідних аргументів на рисунку 2.19 буде успішно пройдена і mmap успішно виконається із розміром 0xfffa000, і зловмисний процес буде мати доступ до пам'яті за межами передбаченого буфера.

Ці приклади не зовсім то і штучні. На рисунку 2.20 наведено інший приклад із вразливим кодом, але вже цей код взято із реального драйверу Linux. Дослідники знайшли цю вразливість досить наївних способом, використовуючи утиліту grep із значенням пошуку remap_pfn_range(. Це ще раз доводить, що ці вразливості існують і їх не дуже важко знайти.

Щодо наведеного коду, то це класичний приклад для цілочисельного переповнення (integer overflow), як і в попередньому прикладі.

```
static int udl_fb_mmap(struct fb_info *info, struct vm_area_struct *vma)
{
    unsigned long start = vma->vm_start;
    unsigned long size = vma->vm_end - vma->vm_start;
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    unsigned long page, pos;

    // EI: offset + size can Integer-Overflow to pass the size limits
    // EI: 1) offset = 2 ** 64 - info->fix.smem_start + <wanted address>
    // EI: 2) size   = wanted size
    // EI: ➔ We can map lower physical pages to user-space
    if (offset + size > info->fix.smem_len)
        return -EINVAL;

    pos = (unsigned long)info->fix.smem_start + offset;

    pr_notice("mmap() framebuffer addr:%lu size:%lu\n",
          pos, size);

    /* We don't want the framebuffer to be mapped encrypted */
    vma->vm_page_prot = pgprot_decrypted(vma->vm_page_prot);

    while (size > 0) {
        page = vmalloc_to_pfn((void *)pos);
        if (remap_pfn_range(vma, start, page, PAGE_SIZE, PAGE_SHARED))
            return -EAGAIN;

        start += PAGE_SIZE;
        pos += PAGE_SIZE;
        if (size > PAGE_SIZE)
            size -= PAGE_SIZE;
        else
            size = 0;
    }

    /* VM_IO | VM_DONTEXPAND | VM_DONTDUMP are set by remap_pfn_range() */
    return 0;
}
```

Рисунок 2.20 – CVE 2018-8781[15]

2.3 Rootkit

Основним завданням вторгнення в інформаційну систему є отримання привілейованого доступу та підтримувати його. Під rootkit розуміється утиліта, яку встановлює зловмисник у зламану систему, зазвичай це сніфери, сканери, троянські програми, які заміняють основні утиліти системи, і що найважливіше, вони використовуються зловмисником після отримання несанкціонованого доступу до системи, наприклад проексплоітував наведені раніше вразливості. Rootkit має три основні функції:

- підтримувати доступ до скомпрометованої системи;
- атакувати інші системи;
- приховати докази діяльності зловмисника.

Існує принаймні три види руткітів. Першим і найпростішим видом є бінарні руткіти, які є звичайними бінарними файлами (наприклад, троянський застосунок). Друга категорія включає бібліотечні руткіти - троянські системні бібліотеки, які розміщені у системі. Ці дві категорії відносно легко виявити: або вручну перевіряючи файлову систему або за допомогою статично пов'язаних файлів. Третя і найбільш підступна категорія руткітів являє собою руткіти ядра системи. Існує дві підкатегорії руткітів ядра:

- завантажуваний руткіт (LKM)
- руткіти з патчем ядра, що безпосередньо змінює образ пам'яті в /dev/mem[18].

Руткіти на рівні ядра зазвичай змінюють таблицю системних викликів. Можна виділити такі три види атак:

- Модифікація таблиці системних викликів: атакуючий модифікує певні адреси в таблиці системних викликів, щоб вказати на нові, шкідливі системні виклики.
- Модифікація цільових системних викликів: перезапис санкціонованих функцій по адресі в системній таблиці викликів, не змінюючи таблицю системних викликів. Тобто, кілька інструкцій функцій системного виклику перезаписуються, і викликаючий процес викличе модифіковану функцію із шкідливим кодом.
- Перенаправлення таблиці системних викликів: руткіт переспрямовує всі посилання до легітимної таблиці системних викликів на нову зловмисно сформовану таблицю за новою адресою.

Розглянемо приклад першого типу атак. Тобто ми знайдемо адресу таблиці

системних викликів та замінено одну функцію на нашу. На рисунку 2.21 зображено код руткіту, який змінює таблицю системних викликів.

```
14  asmlinkage long (*real_chdir)(const char __user *filename);
15
16  unsigned long *syscall_table = (unsigned long*)0xffffffff9e600240;
17
18  asmlinkage long chdir_patch(const char __user *filename)
19  {
20      printk("Patchded chdir is executed!\n");
21      return (*real_chdir)(filename);
22  }
23
24  int __init chdir_init(void)
25  {
26      unsigned int l;
27      pte_t *pte;
28      pte = lookup_address((unsigned long)syscall_table, &l);
29      pte->pte |= _PAGE_RW;
30      real_chdir = (void*)*(syscall_table + __NR_chdir);
31      *(syscall_table + __NR_chdir) = (unsigned long)chdir_patch;
32      printk("Patched!\nOLD :%p\nIN-TABLE:%p\nNEW:%p\n", real_chdir, syscall_table[__NR_chdir], chdir_patch);
33      return 0;
34  }
35
36  void __exit chdir_cleanup(void)
37  {
38      unsigned int l;
39      pte_t *pte;
40      *(syscall_table + __NR_chdir) = (unsigned long)real_chdir;
41      pte = lookup_address((unsigned long)syscall_table, &l);
42      pte->pte &= ~_PAGE_RW;
43      printk("Cleanup,exit\n");
44      return 0;
45  }
46
47  module_init(chdir_init);
48  module_exit(chdir_cleanup);
49  MODULE_LICENSE("GPL");
```

Рисунок 2.21 – Приклад коду руткіта

Даний руткіт виконано у вигляді динамічно завантажуємого модулю (LKM). Про це свідчать функції ініціалізації (module_init) та завершення (module_exit). Module_init є функцією ініціалізації модуля, яка виконується при його першому завантаженні. Ключове слово __init вказує ядру, що цей код буде виконаний один разу, коли модуль завантажиться. Макрос module_init() повідомляє ядру, яку функцію виконати при завантаженні модуля. Все інше, що відбувається всередині ядра – результат установок функції ініціалізації модуля. Так само функція виходу запускається один раз, коли модуль вивантажується, а макрос module_exit() ідентифікує функцію виходу. Ключове слово __exit вказує ядру, що цей код потрібно виконати одного разу, під час вивантаження модуля. За завантаження модулю відповідає консольна команда insmod, а за вивантаження – rmmod,

подивитися на всі завантажені модулі можна за допомогою команди lsmod.

На 16й строчці на рисунку 2.21 оголошена змінна syscall_table. Значення адреси таблиці системних викликів було знайдено за допомогою команди - *cat /proc/kallsyms | grep sys_call_table (рисунок 2.22)*

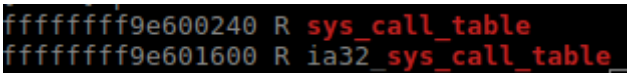

Рисунок 2.22 – Результат виклику наведеної команди

На рисунку 2.22 можна побачити результат виклику цієї команди. Ця команда показало адресу таблиці системних викликів, а також права доступу до неї. Нажаль для зловмисника, вона має лише права на читання, але це можна виправити, далі буде показано як. На 18 строчці зображено визначення нового системного виклику, в цьому прикладі це функція chdir (вона викликається, коли визивається консольна команда cd). Окрім виклику оригінальної функції, в новій функції також викликається функція виводу логу printk, щоб можна було зрозуміти, що перевизначення вдалося. Далі на строчці 24 наведено визначення функції chdir_init, що буде викликатися при ініціалізації модулю. За допомогою функції lookup_address, у функції ініціалізації модуля ми отримали page table entry сторінки, на якій знаходиться таблиця системних викликів, і змінили її атрибути доступу на RW (read | write). Тепер стало можливим змінювати таблицю системних викликів. На строчці 31 відбувається підміна реальної функції chdir на змінену chdir_patch.

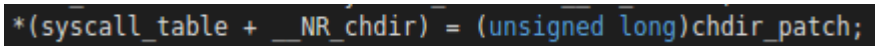

Рисунок 2.23 – Підміна функції.

__NR_chdir – це значення відступу у таблиці системних викликів, від її начальної адреси до реальної функції __NR_chdir, тобто зміщення syscall_table + __NR_chdir вкаже на реальну адресу функції chdir.

На 36 строчці відображено визначення функції chdir_cleanup, ця функція буде викликана при вивантаженні модуля. В цій функції повертаються попередні значення таблиці системних викликів, а також права доступу до таблиці.

Після завантаження модулю і подальших викликів команди cd та rmmod, в логах dmesg можна побачити наступне (рисунок 2.24 - 2.26):

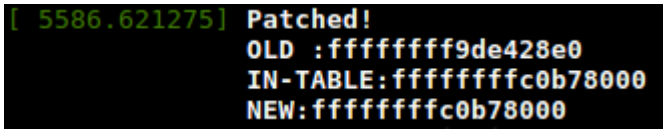

Рисунок 2.24 – Логи dmesg після завантаження модулю

Рисунок 2.25 – Логи dmesg після виконання команди cd

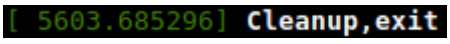

Рисунок 2.26 – Логи dmesg після вивантаження модулю

Отже маючи root привілеї, зловмисник здатен завантажити свій модуль, який може підмінити таблицю системних викликів. Це дасть зловмиснику змогу слідкувати за діями користувача, збирати додаткову інформацію про нього, та навіть більше, якщо зловмисник зможе успішно встановити свій руткіт у систему жертви, то він зможе контролювати всі процеси в просторі користувача і навіть у просторі ядру. Більш складні руткіти можуть приховувати себе від аналізаторів зловмисного програмного забезпечення, або навіть вимкнути їх.

Руткіти є загальною проблемою для звичайних операційних систем, які є великими, складними програмними системами, і можуть містити багато прихованих вразливостей. Працюючи із багатьма процесами з повними привілеями, система буде дуже вразливою, і в результаті компрометації іноді

буває дуже легко встановити руктіт.

Задача, яка розглядається в даній роботі, полягає в тому, чи можна використовувати наявні апаратні засоби смартфонів, щоб обмежити та, можливо, виявити руткіти, які вже введені в працюючий ядро операційної системи мобільного пристрою.

2.4 Аналіз задачі виявлення руткітів.

Для обмеження роботи або виявлення руткітів, дослідники запропонували використовувати апаратні та програмні механізми, такі як віртуалізація та безпечні співпроцесори, які контролюють саму операційну систему. Використовуючи такі механізми, можливо впливати на фактичний стан процесора, а також з’являється можливість моніторингу активності операційної системи.

Гіпервізор може контролювати доступ операційних систем до апаратних ресурсів, таких як пам'ять, а ще може впроваджувати програмне забезпечення для виявлення вторгнення.

Проте зловмисна програма здатна визначити, що вона знаходиться у віртуальному середовищі, завдяки збору артефактів цього середовища і, встановивши, що вона аналізується - перестає виконувати будь-яку зловмисну дію. Певні дослідження доводять, що емуляцію або віртуалізацію можна легко виявити за допомогою певних відбитків, таких як певні рядки в пам’яті або відсутність певних апаратних компонентів і сповільнення продуктивності.

Дослідники вірусних програм [4] визначили методи протидії аналізу, які можуть використовуватися застосунками Android для уникнення виявлення. Вони їх поділили на три категорії:

- статична евристика, заснована на статичній інформації, яка завжди ініціалізується до фіксованих значень у емуляційному середовищі;
- динамічної евристики, заснованої на спостереженні нереалістичної поведінки різних датчиків;

- гіпервізора евристика, заснована на неповній емуляції фактичного обладнання.

Статична евристика полягає у виявленні емулюваного середовища, перевіряючи наявність та вміст унікальних ідентифікаторів пристрою, таких як серійний номер (ідентифікатор пристрою), поточну версію збірки або макет таблиці маршрутизації. Евристика базується на перевірці цих ідентифікаторів, наприклад, чи є IMEI дорівнює нулю, що відповідає стандартній конфігурації Android Emulator.

Динамічна евристика полягає в тому, що мобільні телефони оснащені різними датчиками, в тому числі акселерометр, гіроскоп, GPS, гравітаційний датчик тощо. По суті, ці датчики виводять значення на основі інформації, зібраної з навколишнього середовища, і, таким чином, імітуючи їх реалістично, є складним завданням. Існування сенсорів є ключовою різницею між смартфонами та звичайними обчислювальними системами. Зростаюча кількість датчиків на смартфонах дає нові можливості для ідентифікації фактичних мобільних пристроїв, а отже, і для виявлення емуляторів. За умовчанням емулятор Android не може імітувати рух пристроїв; це може бути досягнуто лише за рахунок додаткових симуляторів датчиків.

З цього можна зробити висновок, що технологія віртуалізації не досить ефективна для аналізу такого зловмисного програмного забезпечення. Також вона може сповільнювати роботу мобільного пристрою.

Щодо антивірусних програм, які також існують і для ринку мобільних пристроїв, то вони також не є ефективними. Вони так само, як і емулятори можуть залишати певні артефакти, які можуть бути виявлені. Також вони працюють на одному привілейованому рівні із зловмисним програмним забезпеченням, а іноді і на нижчому рівні. Руткіт має змогу вимкнути антивірус, або навіть проексплуатувати його вразливості, якщо такі існують.

Більш того, антивірусні програми погано справляються з новими видами вірусів, оскільки вони можуть не мати схожої сигнатури.

Отже визначивши це, в даній роботі пропонується інший підхід для аналізу

зловмисного програмного забезпечення для мобільних пристроїв. Він базується на існуючих рішеннях та розширює їх функціонал.

2.5 Огляд існуючих рішень

Дана робота спирається на такі існуючі рішення, як kprobes[25] та sprobes[23]. Kprobes дозволяє динамічно втрутитися в будь-яку програму ядра та збирати інформацію про неї без рекомпіляції ядра Linux.

Коли реєструється kprobe, kprobes робить копію досліджуваної інструкції і замінює її перший байт інструкції з breakpoint інструкцією (наприклад, int3 на i386 та x86_64).

Коли процесор отримує breakpoint інструкцію, трапляється зупинка, регістри процесора зберігаються, а керування передає Kprobes за допомогою механізму notifier_call_chain. Kprobes виконує pre_handler, який пов'язаний з реєстрованою kprobe, передаючи обробнику адреси структури kprobe і збережені регістри.

Далі Kprobes одноразово виконує копію досліджуваної інструкції.

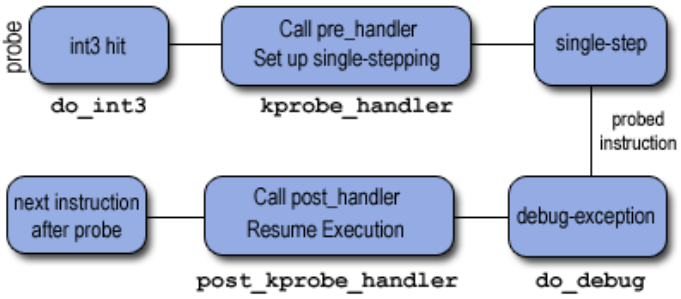

Рисунок 2.27 – Цикл роботи kprobes

Після того, як досліджувана інструкція виконалась, Kprobes виконує post_handler. Потім продовжується виконання наступної за досліджуваною інструкції (рисунок 2.27).

Отже сутність роботи kprobes полягає в перехопленні функції та зміни

певних інструкцій, щоб їх можна було дослідити.

Для перехоплення функцій виконується спеціальна модифікація коду ядра таким чином, щоб забезпечити можливість передачі управління на функцію-перехоплювач при виклику цільової функції. При цьому, існує безлічі варіантів зміни потоку виконання (наприклад, команда jmp).

Таким чином, kprobes модифікує пролог цільової функції таким чином, щоб її виконання процесором призводило до виключення, обробка якого була б контрольованою krpobes. Тобто для кожної цільової функції здійснюється модифікація прологу шляхом запису в її початок команди виникнення винятку, наприклад int 3 з попереднім налаштуванням оброблювачів винятків для всіх таких адрес.

Приклад використання kprobes для функції do_fork. Для того, щоб встановити probe для функції do_fork, потрібно визначити структуру kprobe із функцією, яку будем досліджувати.

```
static struct kprobe kp = {
        .symbol_name    = "do_fork",
};
```

Рисунок 2.28 – Визначення структури kprobe

Далі визначається функція, яка буде виконуватися перед probe. На рисунку 2.29 зображено приклад pre_handler, який виведе логи перед обробкою виключення probe. На рисунку 2.30, так само як і для pre_handler визначається функція post_handler, яка виконається після виконання виключення probe. Якщо під час виконання probe, або pre- чи post_handler виникне виключення, то буде виконуватися fault_handler (рисунок 2.31).

Kprobes дозволяє втрутитися майже в будь-яку частину коду ядра Linux і навіть дозволяє змінювати потій його виконання. Отже це є дуже корисним інструментом для аналізу роботи ядру Linux.

```
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
#ifdef CONFIG_X86
        printk(KERN_INFO "pre_handler: p->addr = 0x%p, ip = %lx,"
                        " flags = 0x%lx\n",
                p->addr, regs->ip, regs->flags);
#endif
#ifdef CONFIG_PPC
        printk(KERN_INFO "pre_handler: p->addr = 0x%p, nip = 0x%lx,"
                        " msr = 0x%lx\n",
                p->addr, regs->nip, regs->msr);
#endif

        /* A dump_stack() here will give a stack backtrace */
        return 0;
}
```

Рисунок 2.29 – Визначення функції, яка виконується перед probe

```
static void handler_post(struct kprobe *p, struct pt_regs *regs,
                                unsigned long flags)
{
#ifdef CONFIG_X86
        printk(KERN_INFO "post_handler: p->addr = 0x%p, flags = 0x%lx\n",
                p->addr, regs->flags);
#endif
#ifdef CONFIG_PPC
        printk(KERN_INFO "post_handler: p->addr = 0x%p, msr = 0x%lx\n",
                p->addr, regs->msr);
#endif
}
```

Рисунок 2.30 – Визначення функції, яка виконується після probe

```
static int handler_fault(struct kprobe *p, struct pt_regs *regs, int trapnr)
{
        printk(KERN_INFO "fault_handler: p->addr = 0x%p, trap #%dn",
                p->addr, trapnr);
        /* Return 0 because we don't handle the fault. */
        return 0;
}
```

Рисунок 2.31 – Визначення функції, яка виконується у разі виключення під час виконання probe.

Щодо sprobes, то це є технікою посилення ядра Linux від rootkit. Розробники sprobes визначили 5 випадків, при яких можлива успішна робота rootkit, та впровадили свої обробники в код ядру Linux, для обмеження роботи rootkit по цім напрямкам.

Наприклад, в ARM існує регістр SCTRL, який відповідає за роботу MMU та WNX механізм захисту пам’яті. Розробники sprobes визначили всі місця в коді ядра Linux, де відбувається обробка цього регістру та встановили туди свої обробники, тобто замість команди зміни відповідного поля регістру SCTRL – викликався обробник sprobes та аналізував команду, яка планувала змінити регістр SCTRL.

Ця техніка використовує технологію trustzone, тому вона може використовуватися лише для ARM пристроїв. Обробник sprobes представляє собою набір команд, який зберігає всі значення регістрів у nwd, для того, щоб їх можна було дослідити в swd. Після збору цих значення формується виклик secure monitor, він виконується завдяки команді SMC 0.

Якщо аналіз команди в swd показав, що її не можна використовувати, наприклад, тому, що вона змінює значення регістру SCTRL, яке відповідає за роботу MMU, то ця команда відкидається і далі не виконується, а користувач отримує попередження.

2.6 Цілі посилення захисту

Метою захисту системи полягає в обмеженні варіантів нападу, доступних зловмиснику. В попередніх розділах було наведено, що операційні системи впроваджують кілька захисних механізмів, таких як W^X і ASLR. Вони запобігають використанню противником внесеного коду в рамках атаки та збільшують складність експлуатації, шляхом зміни адрес розташування стеку, купи тощо. Зловмиснику буде важче визначити місця розташування свого зловмисного коду.

W^X ускладнює зловмиснику експлуатації такої вразливості як buffer

overflow, оскільки ця техніка захисту обмежує використання пам’яті стеку на запис та виконання.

Проте, навіть за такого обмеження, зловмисник все ще може виконувати свої атаки. Наприклад, як було приведено в попередньому розділі, зловмисник може повторно використовують існуючий код, на основі ідеї Return Oriented Programming.

Проте, зловмисник, який вже встановив руткіт в атаковану системи, може ще більше послабити її захист. Оскільки W^X є загальною технікою, яка використовується ядром для запобігання атак, руткіт може просто відключити цей вид захисту.

Для процесорів ARM наприклад, це відбувається шляхом відключення біта Never Execute bit. Коли його встановлено, сторінки пам’яті, в які можна робити записи, вже ніколи не зможуть виконуватися, незалежно від того, як налаштовується таблиця сторінок. Також для обходу W^X, руткіти можуть змінювати записи у таблиць сторінок пам’яті.

Припустимо, що зловмисник хоче змінити сторінку пам’яті з кодом, яка спочатку була доступна лише для виконання. По-перше, йому потрібно змінити біти дозволів цієї сторінки з виконання на запис. Не можливо лише добавити до файлу із правами на виконання право на запис, оскільки ці правила є взаємовиключними. Після зміни виконання на запис, зловмисник може писати на цю сторінку все що йому потрібно. Потім він знов змінює дозвіл цієї сторінки на виконання.

Альтернативою наведеним підходам до експлуатації є дублювання таблиць сторінок в іншому місці та скидання базової адреси таблиці сторінок, в ARM за це відповідає регістр TTBR. Після цього можна виконати попередню техніку. А також зловмисник може просто відключити MMU, і це обійде всі існуючі механізми захисту пам'яті, оскільки всі вони ґрунтуються на системі віртуальної пам'яті.

Виходячи з цього, в якості цілей захисту були обрані MMU та NX bit регістру SCTRL. На щастя, регістр SCTRL також контролює роботу і MMU.

Тому в даному випадку потрібно лише контролювати роботу з даним регістром.

Ці цілі були обрані в якості прикладу. Також в якості цілей посилення можна використовувати всі випадки, які оброблює sprobes.

Висновки до розділу 2

В цьому розділі було розглянуто основні вразливості та приклади їх експлуатації у просторі користувача та просторі ядра. Основними вразливостями як у просторі користувача так і у просторі ядра є buffer overflow, heap overflow, integer overflow, race conditions, memory leaks тощо.

Buffer overflow дозволяє зловмиснику оволодіти керування процесу завдяки перетиранню регістру PC. Після цього зловмисник здатен змінити потік виконання програми, наприклад перевести його на виконання свого зловмисного коду. Для вирішення цієї проблеми існують такі механізми захисту як ASLR та DEP. Але така техніка експлуатації як heap spray здатна обійти ASLR, а також в самому механізмі ASLR існують вразливості, які дають змогу його обійти. Теж саме стосується і DEP. Така техніка експлуатації як Return Oriented Programming дозволяє знайти спеціальні інструкції у пам’яті, в якій дозволено виконувати код. Знайшовши такі інструкції (їх ще називають гаджетами), зловмисник здатен виконати свій код. Наприклад, він може викликати функції mmap та mprotect та виділити собі пам'ять та надати їх прав на запис та виконання, а потом завантажити туди свій код, та виконати його.

Були розглянуті специфічні для простору ядра загрози. Наприклад, зловмисник здатен проексплуатувати певну вразливість в просторі користувача та отримати root привілеї, а після цього завантажити свій rootkit.

Простим прикладом роботи rootkit є перезапис таблиці системних викликів. Для цього зловмисник достатньо дізнатися адреси sys_call_table та визначити певний відступ до функції із цієї таблиці та замінити її на свою.

Також зловмисник може змінити значення регістру SCTLR (для ARM), а саме вимкнути NX bit захист. Після цього зловмисник зможе записувати в пам’ять свій код та виконувати його. Ще одним із способів зловмисного використання є вимкнення MMU, це теж робиться за допомогою наведеного регістру. В даному випадку, це може ускладнити експлуатацію, але якщо у вразливій системі фізичні адреса відображаються прямо у віртуальні без відступів, то це дасть змогу

виконати зловмисний код не турбуючись про механізми захисту пам’яті.

Також були розглянуті такі техніки динамічного аналізу коду, як kprobes та sprobes.

Сутність роботи kprobes полягає в перехопленні функції та зміни її прологу таким чином, щоб можливо було перевести виконання до функції-перехоплювачу при виклику цільової функції. Це дає змогу аналізувати та навіть змінювати потік виконання цільової функції. Таким чином, kprobes модифікує пролог цільової функції таким чином, щоб її виконання процесором призводило до виключення, обробка якого була б контрольованою kprobes.

Sprobes є технікою посилення ядра Linux від rootkit. використовує технологію trustzone, тому вона може використовуватися лише для ARM пристроїв. Обробник sprobes представляє собою набір команд, який зберігає всі значення регістрів у nwd, для того, щоб їх можна було дослідити в swd. Після збору цих значення виконується виклик SMC 0. Розробники sprobes визначили 5 випадків, при яких можлива успішна робота rootkit, та впровадили свої обробники в код ядру Linux, для обмеження роботи rootkit по цім напрямкам.

3 ПОСИЛЕННЯ ЗАХИСТУ ЯДРА LINUX

3.1 Розробка техніки посилення захисту ядра Linux від rootkit

В попередніх розділах наводилися приклади методів та технологій які використовуються для аналізу та виявлення зловмисного програмного забезпечення, але вони мають певні недоліки, які дають змогу зловмисним застосункам визначити, що вони працюють в емульованому середовищі. Тому в даній роботі пропонується інша техніка виявлення зловмисних застосунків, точніше rootkit, вона використовує технологію trustzone, це є secure extension ARM процесорів, тому дана методика буде працювати лише для ARM пристроїв.

Ця техніка представляє собою аналізатор-монітор, який може збирати дані про завантажений у ядро модуль. Для того, щоб позбутися недоліків попередніх рішень, основна частина цього аналізатору буде знаходитися в swd – захищеному середовищі. Завдяки використанню довіреного середовища виконання, аналізатор працює на іншому рівні, ніж можуть виконуватися зловмисні застосунки, руткіти, а саме на рівні S-EL0. Завдяки цьому руткіти ніяк не може впливати на нього.

Основною ідеєю є встановлення обробників у код завантажуваних модулів без перевантаження ядру. Встановлення обробників відбувається лише у випадку, коли в коді модулю буде знайдено команди, які можуть внести зміни до регістру SCTRL. У випадку, коли їх буде знайдено замість них будуть встановлюватися обробники, в іншому – модуль буде завантажений і працювати в звичайному режимі.

У розділі 1.3 було розглянуто принцип завантаження динамічного модулю у ядро Linux. Основну роботу виконує функція load_module. Вона завантажує elf заголовки і секції та виділяє під них тимчасову пам'ять. Для нас важлива секція, яка зберігає виконавчий код динамічного модулю. Її потрібно проаналізувати та знайти команду, яка працює із регістром SCTRL. Після цього, її потрібно замінити на nwd_handler (рисунок 3.1, ліва частина).

```
NWD:
kernel code {
        struct to_swd;
        do something
        .....
        if (!store_data_to_struct(to_swd) &&
            !hook(to_swd))
                return ABORT_EVENT;
        ....
}
```

```
SWD:
handler code {
        struct from_nwd;
        fill_struct(from_nwd);
        ....
        process_hook(from_nwd){
                if (check(from_nwd))
                        return OK;
                else
                        return ABORT_EVENT;
        }
}
```

Рисунок 3.1 – псевдокод nwd_handler, swd_handler

Nwd_handler зберігає значення регістрів цього середовища, та команду, яка намагається змінити SCTRL регістр, а після цього викликає команду SCM для переходу у режим secure monitor. Для того, щоб викликати правильний обробник в swd частині в регістр r0 передається значення, яке відповідає команді обробника – TZ_MONITOR_CHECK_ NWD_SCTRL_MDF.

Для того, щоб змінити потрібні інструкції на обробник необхідно змінити права доступу до сторінки пам’яті, де зберігається вже скопійований із простору користувача модуль (розділ 1.3), адже сторінки пам'яті ядра, що зберігають код і дані, позначені як read-only і захищені від запису.

Найпростішим рішенням даної проблеми є тимчасове відключення NX bit регістра SCTRL, але це досить небезпечне рішення, оскільки потік в якому виконується відключення захисту може також виконуватися і на інших процесорах, і він також знімне цей захист на них. Щоб відключити NX біт потрібно обнулити 19 біт регістру SCTRL (рисунок 3.2):

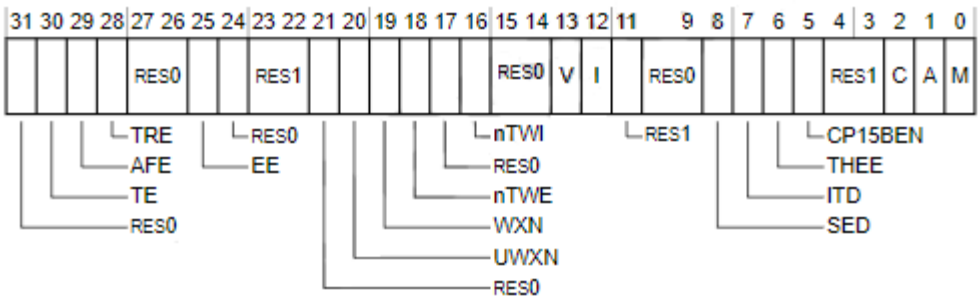

Рисунок 3.2 – Регістр SCTRL

Існує також інший спосіб змінити права доступу сторінки пам’яті - створення тимчасового відображень, тому як дозволено створювати кілька посилань на фізичну пам'ять із різними атрибутами. Для цього можна використати наступний код [26]:

```
static void *map_writable(void *addr, size_t len)
{
	void *vaddr;
	int nr_pages = DIV_ROUND_UP(offset_in_page(addr) + len, PAGE_SIZE);
	struct page **pages = kmalloc(nr_pages * sizeof(*pages), GFP_KERNEL);
	void *page_addr = (void *)((unsigned long)addr & PAGE_MASK);
	int i;

	if (pages == NULL)
		return NULL;

	for (i = 0; i < nr_pages; i++) {
		if (__module_address((unsigned long)page_addr) == NULL) {
			pages[i] = virt_to_page(page_addr);
			WARN_ON(!PageReserved(pages[i]));
		} else {
			pages[i] = vmalloc_to_page(page_addr);
		}
		if (pages[i] == NULL) {
			kfree(pages);
			return NULL;
		}
		page_addr += PAGE_SIZE;
	}
	vaddr = vmap(pages, nr_pages, VM_MAP, PAGE_KERNEL);
	kfree(pages);
	if (vaddr == NULL)
		return NULL;
	return vaddr + offset_in_page(addr);
}
```

Рисунок 3.3 – Функція для створення відображення пам’яті із доступом на запис.

Отже після підміни команди, буде викликатися обробник, який в свою чергу буде викликати secure monitor для зміни середовища виконання.

```
push    {r4-r8, lr}
mov r8, r0
ldm r8, {r0-r7}
smc #0
stm r8, {r0-r7}
pop {r4-r8, pc}
```

Рисунок 3.4 – приклад виклику secure monitor

Аргументи, які нам потрібно перевірити передаються через структуру smc_param (рисунок 3.5):

```
struct smc_param {
        uint32_t a0;
        uint32_t a1;
        uint32_t a2;
        uint32_t a3;
        uint32_t a4;
        uint32_t a5;
        uint32_t a6;
        uint32_t a7;
};
```

Рисунок 3.5 – smc_param

Яка потім зчитується завдяки командам:

mov r8, r0

ldm r8, {r0-r7}

Після цього, виконавчий процес переходить до secure monitor, який передає керування на траслет, який буде виконувати обробку аргументів (рисунок 3.6).

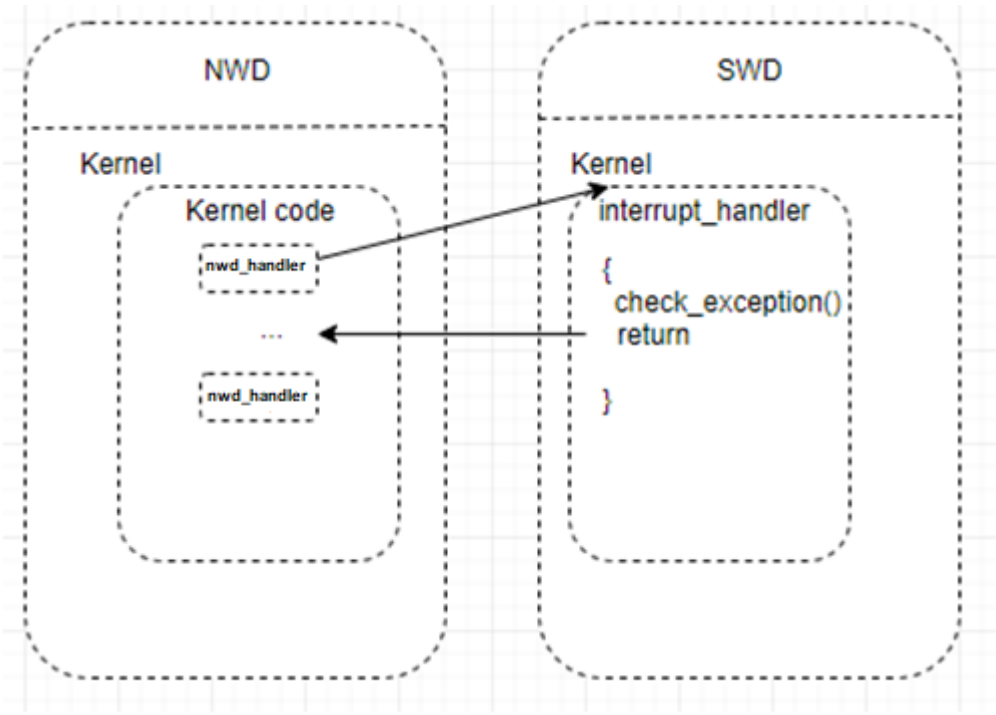

Рисунок 3.6 – Роботи обробника nwd_handler

Тобто, якщо на вхід до функції load_module подається такий модуль (рисунок 3.7), наведеним алгоритмом, спочатку буде його проаналізовано на наявність інструкцій зміни SCTRL регістру, коли такі будуть знайдені (рисунок 3.8), пройде підміна цього коду на код nwd_handler (рисунок 3.4) після чого керування перейде до secure monitor, який викличе обробник в swd (рисунок 3.9).

Отже алгоритм роботи такий:

1. Копіювання модулю із простору користувача та створення тимчасової копії.
2. Парсинг бінарного коду вхідного модулю. Під час парсингу проводиться аналіз інструкцій.
3. Якщо знайдено інструкції, які намагаються отримати доступ до регістру SCTRL, то замінити їх на інструкцію переходу в режим secure monitor, тобто SMC.
4. Завантажити та ініціалізувати досліджуваний модуль.
5. При попаданні на інструкцію SMC, обробити це виключення
6. В безпечному середовищі перевірити команду, яка намагалась

отримати доступ до регістру SCTRL, якщо ці дія зловмисна – заборонити її виконання, та вивантажити модуль.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

static void disable_mmu(void)
{
        u32 fmt = 0;
        asm volatile ("mrc p15, 0, %0, c1, c0, 0" : "=r"(fmt));
}

int __init init(void)
{
        printk("Init, calling disable_mmu test\n");
        disable_mmu();
        return 0;
}

void __exit exit(void)
{
        printk("Cleanup\n");
}

module_init(init);
module_exit(exit);
MODULE_LICENSE("GPL");
```

Рисунок 3.7 – Тестовий модуль

```
Disassembly of section .init.text:

00000000 <init>:
   0:   e92d4008        push    {r3, lr}
   4:   e59f000c        ldr     r0, [pc, #12]   ; 18 <init+0x18>
   8:   ebfffffe        bl      0 <printk>
   c:   ee113f10        mrc     15, 0, r3, cr1, cr0, {0}
  10:   e3a00000        mov     r0, #0
  14:   e8bd8008        pop     {r3, pc}
  18:   00000000        .word   0x00000000

Disassembly of section .exit.text:

00000000 <cleanup_module>:
   0:   e59f0000        .word   0xe59f0000
   4:   eafffffe        b       0 <printk>
   8:   00000020        .word   0x00000020
```

Рисунок 3.8 – Тестовий модуль на асемблері

Відповідно до SMC calling convention (розділ 1.8) у регістрі r0 передається код команди, яка повинна виконатися. У цьому випадку це буде команда TZ_MONITOR_CHECK_ NWD_SCTRL_MDF. З рисунку 3.9 можна побачити, що

функція обробник-команд TA_InvokeCommandEntrypoint як раз очікує виклик цієї команди, отримавши цю команду вона викличе функцію check_sctrl_mdf, яка саме і перевіряє команди із nwd частини. Якщо буде знайдено, що команда планує змінити SCTRL, то вона буде відкликана.

```
static TEE_Result check_sctrl_mdf(uint32_t param_types,
    TEE_Param params[4])
{
    uint32_t exp_param_types =
                TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_OUTPUT,
                        TEE_PARAM_TYPE_NONE,
                        TEE_PARAM_TYPE_NONE,
                        TEE_PARAM_TYPE_NONE);

    DMSG("has been called");
    if (param_types != exp_param_types)
        return TEE_ERROR_BAD_PARAMETERS;

    IMSG("Checking sctlr register");

    if (check_input_command(params[0].memref.buffer,
                            params[0].memref.size))
    {
        IMSG("Abort instruction");
        return TEE_ERROR_BAD_PARAMETERS;
    }

    return TEE_SUCCESS;
}

TEE_Result TA_InvokeCommandEntryPoint(void __maybe_unused *sess_ctx,
            uint32_t cmd_id,
            uint32_t param_types, TEE_Param params[4])
{
    (void)&sess_ctx;

    switch (cmd_id)
    {
        case TZ_TA_MONITOR_CHECK_NWD_SCTRL_MDF:
            return check_sctrl_mdf(param_types, params);
        default:
            return TEE_ERROR_BAD_PARAMETERS;
    }
}
```

Рисунок 3.9 – swd_handler

3.2 Аналіз результатів

У результаті роботи були перевірені 2 наведених тестових випадки, в яких відповідно змінювалися значення регістрів, відповідних за роботу MMU та за NX bit. Спроби зміни регістру SCTRL були виявлені.

Під час імплементації та тестування були виявлені недоліки даної методики. Для аналізу бінарного коду під завантаження модулю необхідно використовувати вже існуючі декомпілятори під ARM платформу, оскільки власна реалізація може мати вразливості в реалізації та погіршувати продуктивність самої системи, що приводить до підвищення часу завантаження динамічного модулю, час завантаження буде помітно вищий для великих модулів. В якості декомпілятора можливо використовувати capstone framework [27] з додатковим змінами, щоб облегшити його розміри.

Також дана технологія не підтримується старими процесорами, оскільки технологія trustzone підтримується для процесорів ARMv6 і вище.

Ще одним недоліком є наявність false-positive помилок, оскільки можливі випадки, при яких модуль дійсно повинен змінювати значення регістру SCTRL. Цей випадок вирішується шляхом впровадження реєстрації санкціонованих модулів, та подальшої їх перевірки. Якщо модуль вже зареєстрований і є довіреним, то його можна запускати без додаткового аналізу та втручання у його бінарний код.

У порівнянні із існуючими рішення, дана технологія не має певних артефактів, які можливо виявити, як це відбувається у випадку використання емуляції. Можливо визначити модель процесору та узнати, що в даному пристрої підтримується trustzone, але вимкнути її не є тривіальною і можливою задачею.

Також потрібні налаштування та методики протидії руткітам, які можуть зашифровувати та розшифровувати свій бінарний код у runtime [28].

На відміну від sprobes, дане рішення не потребує патчу Linux Kernel image та подальшого його розповсюдження клієнтам, які потребують такого типу захисту. Оскільки дана технологія може буде включена відповідним патчем до

Linux Kernel sources там бути використаною одразу після компіляції. Ін’єкція відповідних обробників відбувається динамічно лише після аналізу модулів. Якщо модуль не зареєстрований, то він потребує такого втручання.

Оскільки аналіз відбувається лише для завантажуваних модулів, а не для всього kernel image, то продуктивність системи буде вища ніж для sprobes. Теж саме стосується і false-positive помилок.

Висновки до розділу 3

В даному розділі було запропоноване рішення, яке є дозволяє аналізувати динамічно завантажуванні модулі. Дане рішення використовує технологію довіреного середовища виконання, що дозволяє йому бути не доступним до руткітів.

Дане рішення можна використовувати для аналізу бінарного коду модулів, знаходячись в захищеному середовищі, не залишаючи ніякі артефакти, що дають змогу руткіту упізнати аналізуючу систему.

Пропоноване рішення представляє собою певний Sandbox, в якому працює модуль. Якщо модуль звертається до важливих до інформаційної безпеки ресурсів, то замість інструкцій звернення, наприклад, до регістру SCTRL встановлюється виклик команди SMC, а далі обробник виключень буде перевіряти наведену інструкцію.

Запропоноване рішення було порівняно з типовими методами аналізу руткітів. Результат показав, що технологія TEE має ряд переваг на типовими методами. Оскільки TEE має більш привілейований доступ до ресурсів, то вона може використовуватися як монітор, логер або аналізатор системи. Це ускладнює приховування своєї діяльності для руткітів, оскільки вони не знатимуть, що вони аналізуються.

Недоліками даної технології є складність її реалізації. Це стосується аналізу бінарного коду динамічно завантажуваного модулю. А також платформо-залежність. Даний механізм можна лише використовувати під ARM платформу і не підтримується старими процесорами, оскільки технологія trustzone підтримується для процесорів ARMv6 і вище.

Ще одним недоліком є наявність false-positive помилок, оскільки можливі випадки, при яких модуль дійсно повинен змінювати значення регістру SCTRL. Цей випадок вирішується шляхом впровадження реєстрації санкціонованих модулів, та подальшої їх перевірки. Якщо модуль вже зареєстрований і є

довіреним, то його можна запускати без додаткового аналізу та втручання у його бінарний код.

Запропоноване рішення може використовуватися для впровадження політик роботи модулів ядра, а також може використовуватися в якості аналізатору зловмисних застосунків.

4 РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ

4.1. Опис ідеї проекту

Таблиця 4.1 - Опис ідеї стартап-проекту

Зміст ідеї	*Напрямки застосування*	*Вигоди для користувача*
Моніторинг безпеки операційної системи. Аналіз зловмисного програмного забезпечення, впровадження політик доступу для застосунків	1. Виробники одноплатних компьютерів, смартфонів	Загортовування безпеки операційної системи для своїх виробів
	2. Звичайні користувачі мобільними пристроями, поставляється в якості додаткового сервісу безпеки	Отримання додаткового інструмента керування безпекою мобільного пристрою.
	3. Дослідникам зловмисного програмного забезпечення	Ефективний інструмент для аналізу зловмисного програмного забезпечення, який є не видимим для зловмисного ПО.

Таблиця 4.2 - Визначення сильних, слабких та нейтральних характеристик ідеї проекту

1	*2*	*3*		*4*	*5*	*6*
№ п/п	*Техніко-економічні характеристики ідеї*	*(потенційні) товари/концепції конкурентів*		*W (слабка сторона)*	*N (нейтральна сторона)*	*S (сильна сторона)*
		Мій проект	*Виробники антивірусів*			
1.	економічні	Витрати на розробку рішення, – 6000 $	Витрати на розробку рішення, закупку ліцензій, сервіс та маркетинг - 250,000 $	Недостатній рівень витрат на маркетинг та рекламу	Схожий функціонал	Дешевша реалізація проекту. Переваги в аналізі, та прозорості для зловмисного програмного забезпечення
2.	технічні	Використання сучасних технологій в індустрії мобільної безпеки -trustzone	Складні алгоритми аналізу поведінки	Складність у розробці програмного забезпечення	немає	Більша кількість сфер використання рішення для забезпечення інформаційної безпеки

Продовження таблиці 4.2.

1	2	3		5	6	7
3.	технологічні	Використання платформ, що надають інформацію про загрози, індикатори компрометації, дисасемблер	Відсутність таких компонентів	немає	Більші видатки на експлуатацію рішення	Отримання більшої кількості інформації для підвищення ефективності роботи SOC та якості надаваних послуг
4.	ергономічні	Можливість налаштування аналізатору на потрібну задачу	Лише для виявлення зловмисного програмного забезпечення	Додаткова складність при налаштуванні	немає	Різноманітність способів використання
5	органолептичні	Швидкість роботи залежить лише від мобільного пристрою клієнта	Швидкість роботи залежить лише від мобільного пристрою клієнта	немає	Потребує оновлення бази сигнатур	Не потребує оновлення бази сигнатур

4.2. Технологічний аудит ідеї проекту

Таблиця 4.3 - Технологічна здійсненність ідеї проекту

№ п/п	*Ідея проекту*	*Технології її реалізації*	*Наявність технологій*	*Доступність технологій*
1	Динамічний аналіз застосунків	Використання технології trustzone для налаштування прозорого аналізу, потребує змін до ядру Linux	Потрібно придбати ліцензію на використання пропрієтарної tee	Так, ця технологія є доступною
2	Впровадження політик роботи застосунків	Використання технології trustzone для впровадження політик, потребує змін до ядру Linux	Потрібно придбати ліцензію на використання пропрієтарної tee, час на разробку	Так, ця технологія є доступною
3	Логування роботи інформаційної системи	Використання технології trustzone для впровадження політик, потребує змін до ядру Linux	Потрібно придбати ліцензію на використання пропрієтарної tee, час на разробку	Так, ця технологія є доступною
Обрана технологія реалізації ідеї проекту: так як для реалізації ідеї проекту, всі технології є наявними та доступними, тому обираються всі вище описані технології.				

4.3. Аналіз ринкових можливостей запуску стартап-проекту

Таблиця 4.4 - Попередня характеристика потенційного ринку стартап-проекту

1	*2*	*3*
№ п/п	*Показники стану ринку аналізу зловмисного ПО*	*Характеристика*
1	Кількість головних гравців, од	ESET, AVAST, SYMANTEC
2	Загальний обсяг продаж, грн/ум.од	3 млн ум.од
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу	Потреба у кадрах із високим рівнем компетентності у сфері інформаційної безпеки
5	Специфічні вимоги до стандартизації та сертифікації	немає
6	Середня норма рентабельності в галузі, %	Не менше 100

Таблиця 4.5 - Характеристика потенційних клієнтів стартап-проекту

№ п/п	*Потреба, що формує ринок*	*Цільова аудиторія (цільові сегменти ринку)*	*Відмінності у поведінці різних потенційних цільових груп клієнтів*	*Вимоги споживачів до товару*
1	Аналіз зловмисного програмного забезпечення на мобільних пристроях Моніторинг, аудит безпеки інформаційних систем мобільних пристроїв	- Виробники одноплатних компьютерів, смартфонів - Звичайні користувачі мобільними пристроями, - Дослідники зловмисного програмного забезпечення	Для кожної категорій існують окремі цінності пропонованого рішення, наприклад виробникам одноплатних компьютерів важливо вдосконалити безпеку операційної системи свого продукту. Звичайні користуві прагнуть зберігати свій мобільний пристрій у безпеці від вірусів та іншого зловмисного програмного забезпечення. Для дослідників вірусів важливо отримати ісінтрумент для аналіз зловмисного ПО	- ефективний аналіз зловмисних дій застосунків. - Швидке та ефективне виявлення зловмисного програмного забезпечення

Таблиця 4.6 - Фактори загроз

№ п/п	*Фактор*	*Зміст загрози*	*Можлива реакція компанії*
1	Цінова конкуренція	Демпінг цін на послуги конкурентів	Зниження вартості продукту
2	Зниження доходів потенційних споживачів	Зниження купівельної спроможності клієнтів	Зниження вартості продукту

Таблиця 4.7 - Фактори можливостей

№ п/п	*Фактор*	*Зміст можливості*	*Можлива реакція компанії*
1	Поява нових технологій	Розширення спектру надаваних послуг для клієнтів, збільшення ефективності роботи сервісу,	Швидке впровадження нових технологій, створення нових сервісів
2	Залучення нових відомих компаній до	Розширення використання	Розробка та впровадження нових засобів та програмних

	співробітництва	пропонованого продукту	комплексів.

Таблиця 4.8 - Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	*В чому проявляється дана характеристика*	*Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)*
1. олігополістична конкуренція	Динаміка цін, яка майже не залежить від рівню попиту на продукцію;	Вдосконалення рішення для підвищення якості інструменту
2. За рівнем конкурентної боротьби - національний	Надання інструменту для різних груп та типів клієнтів по всіму світу	Застосування нових технологій та підходів для вдосконалення рішення
3. За галузевою ознакою - міжгалузева	Рішення може використовуватися користувачами з різних галузей	Впровадження нових функцій сервісу
4. Конкуренція за видами товарів: - товарно-видова	Рішення, що використовується для задоволення потреб клієнтів, але технологічно відрізняються від рішень конкурентів	Надання інструменту для аналізу зловмисного програмного забезпечення
5. За характером конкурентних переваг - цінова	Цінова	Надання ширшого функціоналу інструменту
6. За інтенсивністю - марочна	Сукупність характеристик та властивостей рішення	Підвищення якості роботи інструменту

Таблиця 4.9 - Аналіз конкуренції в галузі за М. Портером

	Прямі конкуренти в галузі	*Потенційні конкуренти*	*Постачальники*	*Клієнти*	*Товари-замінники*
Складові аналізу	ESET, AVAST, SYMANTEC.	Розмір капіталовкладень; доступ до ресурсів у конкурентів; наявність патентів та товарних знаків.	немає	Розміри закупівель державних підприємств та органів влади, силових структур;	Ціна товару та змінні витрати
Висновки:	Наявна досить інтенсивна конкурентна боротьба з боку прямих	Є можливості входу на ринок, але існує багато конкурентів	немає	Клієнти диктують умови на ринку; при організації	немає.

	конкурентів			закупівель товарів та послуг	

Даний проект має принципові можливості для роботи на ринку з огляду на конкурентну ситуацію. Серед сильних сторін, можна виділити використання нових технологій в мобільній сфері, надання інструменти із широким функціоналом; залучення великих відомих компаній в якості постачальників чи клієнтів.

Таблиця 4.10 - Обґрунтування факторів конкурентоспроможності

№ п/п	*Фактор конкурентоспроможності*	*Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)*
1	Ціна	Ціноутворення, яке є однаковим для різних клієнтів
2	Гнучкість цін на продукт	Є можливість зменшення цін на продукт
3	Різномантіний фнкціонал	Різноманітність функціоналу значно ширше, в порівнянні з конкурентами

Таблиця 4.11 - Порівняльний аналіз сильних та слабких сторін «Довірений монітор»

№ п/п	*Фактор конкурентоспроможності*	*Бали 1-20*	*Рейтинг товарів-конкурентів у порівнянні з довірений монітор*						
			–3	*–2*	*–1*	*0*	*+1*	*+2*	*+3*
1	Ціна	14			+				
4	Гнучкість цін на послуги	13		+					
5	Різномантіний фнкціонал	16		+					

Таблиця 4.12 - SWOT-аналіз стартап-проекту

Сильні сторони: різномантіний фнкціонал продукту; ціноутворення, яке є однаковим для різних типів клієнтів	*Слабкі сторони*: низький рівень маркетингу;
Можливості: швидке впровадження нових технологій залучення великих відомих компаній у співробітництві	*Загрози*: цінова конкуренція;зниження доходів потенційних споживачів;

Таблиця 4.13 - Альтернативи ринкового впровадження стартап-проекту

№ п/п	*Альтернатива (орієнтовний комплекс заходів) ринкової поведінки*	*Ймовірність отримання ресурсів*	*Строки реалізації*
1	Побудова динамічної аудит-системи із логуванням	Не потрібно	3 місяці

З означених альтернатив ринкового впровадження даного стартап-проекту було

вирішено обрати побудову динамічної аудит-системи із логуванням, при цьому строки реалізації становитимуть приблизно 3 місяці.

4.4. Розроблення ринкової стратегії проекту

Таблиця 4.14 - Вибір цільових груп потенційних споживачів

1	*2*	*3*	*4*	*5*	*5*
№ п/п	*Опис профілю цільової групи потенційних клієнтів*	*Готовність споживачів сприйняти продукт*	*Орієнтовний попит в межах цільової групи (сегменту)*	*Інтенсивність конкуренції в сегменті*	*Простота входу у сегмент*
1	Виробники одноплатних компьютерів, смартфонів	Клієнти потребують продукт такого типу та готові ним користуватись	Високий попит, пов'язаний із необхідністю послуг у сфері кібербезпеки	Середній рівень конкуренції	немає
2	Звичайні користувачі мобільними пристроями	Клієнти зацікавлені продуктом	Середній рівень попиту	Середній рівень конкуренції	немає
3	Дослідникам зловмисного програмного забезпечення	Клієнти зацікавлені продуктом	Середній рівень попиту	Середній рівень конкуренції	немає

На підставі ринкової стратегії обрано використання стратегії диференційованого маркетингу.

Таблиця 4.15 - Визначення базової стратегії розвитку

№ п/п	*Обрана альтернатива розвитку проекту*	*Стратегія охоплення ринку*	*Ключові конкурентоспроможні позиції відповідно до обраної альтернативи*	*Базова стратегія розвитку*
1	Флангова атака	Стратегія диференційованого маркетингу	Сильні сторони та можливості рішення	Стратегія диференціації

Таблиця 4.16 - Визначення базової стратегії конкурентної поведінки

№ п/п	*Чи є проект «першопрохідцем» на ринку?*	*Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?*	*Чи буде компанія копіювати основні характеристики товару конкурента, і які?*	*Стратегія конкурентної поведінки*

1	Ні	Так	Ні	Стратегія виклику лідера

Таблиця 4.17 - Визначення стратегії позиціонування

№ п/п	*Вимоги до товару цільової аудиторії*	*Базова стратегія розвитку*	*Ключові конкурентоспроможні позиції власного стартап-проекту*	*Вибір асоціацій, які мають сформувати комплексну позицію власного проекту*
1	Отримання єдиного інструмента керування безпекою та комплексного бачення стану захищеності, забезпечення інформаційної безпеки. Забезпечення захищеності інформаційної структури.	Стратегія диференціації	Сильні сторони та можливості рішення	Моніторинг безпеки інформаційних систем операційної системи мобільних пристроїв клієнтів

4.5. Розроблення маркетингової програми стартап-проекту

Таблиця 4.18 - Визначення ключових переваг концепції потенційного товару

1	*2*	*3*	*4*
№ п/п	*Потреба*	*Вигода, яку пропонує товар*	*Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити*
1	Моніторинг, аудит безпеки інформаційних систем мобільних пристроїв	Забезпечення захищеності інформаційної структури	Отримання єдиного інструмента керування безпекою та комплексного бачення стану захищеності. Швидке та ефективне виявлення зловмисного програмного забезпечення
2	Аналіз зловмисного програмного забезпечення на мобільних пристроях	ефективний аналіз зловмисних дій застосунків.	аналітика

Таблиця 4.19 - Опис трьох рівнів моделі товару

Рівні товару	*Сутність та складові*		
I. Товар за задумом	Моніторинг безпеки операційної системи. Аналіз зловмисного програмного забезпечення, впровадження політик доступу для застосунків.		
II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх /Тл/Е/Ор
	1. Використання trustzone	М	Тх/Тл/Е
	2. Оновлення та виправлення для програмного забезпечення	Нм	Вр/Тл/Е
	4. Можливість налаштування логування	М	Тх/Е/Ор
	5. Швидкість роботи рішення для користувача	Нм	Тл/Е/Ор
	Якість: Забезпечення захищеності інформаційної структури.		
	Надання користувачу можливості встановленя політик доступу, логування подій		
	Марка: довірений монітор		
III. Товар із підкріпленням	До продажу: для стимулювання попиту на продукт можна розробити програму надання тимчасового доступу до рішення.		
	Після продажу: Розробка та проведення рекламної кампанії		
За рахунок чого потенційний товар буде захищено від копіювання: захист інтелектуальної власності			

Таблиця 4.20 - Визначення меж встановлення ціни

№ п/п	*Рівень цін на товари-замінники*	*Рівень цін на товари-аналоги*	*Рівень доходів цільової групи споживачів*	*Верхня та нижня межі встановлення ціни на товар/послугу*
1	15000 ум.од	450 ум.од	Високий або середній	Ціна на ліцензію– 350-700 ум.од.;

Таблиця 4.21 - Формування системи збуту

№ п/п	*Специфіка закупівельної поведінки цільових клієнтів*	*Функції збуту, які має виконувати постачальник товару*	*Глибина каналу збуту*	*Оптимальна система збуту*
1	прийняття рішення про необхідність систем моніторингу та керування безпеки, вибір джерела	пристосування збутової мережі до запитів споживачів; пошук перспективних засобів просування	Дворівневий канал збуту (із залученням посередника та дистриб'ютора)	Залучення компаній посередників та партнерів для формування системи

	задоволення своїх потреб для забезпечення безпеки інформаційних систем і укладення угоди	товарів; розробка та вдосконалення маркетингової політики; вибір посередників		збуту

Таблиця 4.22 - Концепція маркетингових комунікацій

№ п/п	*Специфіка поведінки цільових клієнтів*	*Канали комунікацій, якими користуються цільові клієнти*	*Ключові позиції, обрані для позиціонування*	*Завдання рекламного повідомлення*	*Концепція рекламного звернення*
1	Дослідження властивостей та якостей рішення, можливість тестування продукту, прийняття рішення про необхідність використання продукту для задоволення своїх потреб	Канали інтегрованих маркетингових комунікацій	Використання сильних сторін рішення: trustzone; надання широкого функціоналу;швидке впровадження нових технологій	повідомлення, пов’язані з особистою вигодою аудиторії, що показують, як товар може задовольняти потреби покупця;	реклама, що демонструє якість товару, його економічність, цінність або можливості експлуатації

Висновки до розділу 4.

Результатом проведеного аналізу та оцінки ризиків, було виявлено, що даний проект має можливість для ринкової комерціалізації. Для представленого рішення є високий рівень попиту, що пов’язаний із станом інформаційної безпеки користувачів, тому робота над проектом має гарну рентабельність на ринку послуг у сфері кібербезпеки. Для цього стартап-проекту є непогані перспективи входження в ринок, але наявні певні бар’єри, подолання яких підвищую конкурентоспроможність представленого рішення. Щодо альтернативи впровадження стартап-проекту та його ринкової реалізації, доцільно обрати механізми для побудови рішення з використанням компонентів однієї компанії постачальника.

ВИСНОВКИ

В даному роботі були розглянуті основні принципи будови операційної системи Linux, основні її функції. Було визначено, що операційна система є програмою, яка управляє всіма іншими програмами, якими користується звичайний користувач, представляє йому графічний інтерфейс. Зі сторони розробника, операційна система це певний інтерфейс, який дозволяє користуватися всіма ресурсами системи. Ядро Linux є важливою частиною Linux-подібної операційної системи. Воно підтримує динамічне завантаження модулів ядра, які можуть розширювати доступний фунціонал ядра Linux. В операційній системі існують механізми захисту пам’яті. Вони ускладнюють експлуатацію вразливостей (buffer overflow, integer overflow тощо).

Були розглянуті таки техніки захисту, впроваджені операційною системою: ASLR та DEP. Принцип роботи ASLR полягає в тому, що він змінює початкові адреси стеку, купи, бібліотек, тому зловмиснику буде важче написати експлоіт. DEP позначає всі сторінки пам’яті, в яких немає коду прапорцем READ. Це не дає змоги виконувати вразливий код зі стеку, який туди помістили, наприклад, після експлуатації вразливості buffer overflow.

Також було розглянуто особливості архітектури ARM. Були розглянуті основні доступні користувачу регістри та приклади їх використання. Було розглянуто технологію Trusted Execution Environment. Основна її суть полягає в тому, що вона поділяє роботу системи на два світи: безпечний та небезпечний. Перемикання між ними виконується завдяки secure monitor, а саме завдяки виклику команди SMC. Однією з її переваг даної технології те, що безпечний світ працює в на іншому рівні привілеїв. І звичайні застосунка та навіть ядро операційної системи не зможуть впливати на безпечне середовище без певних налаштувань.

Було розглянуто основні вразливості та приклади їх експлуатації у просторі користувача та просторі ядра. Основними вразливостями як у просторі користувача так і у просторі ядра є buffer overflow, heap overflow, integer overflow,

race conditions, memory leaks тощо.

Також були розглянуті специфічні для простору ядра загрози. Наприклад, зловмисник здатен проексплуатувати певну вразливість в просторі користувача та отримати root привілеї, а після цього завантажити свій rootkit.

Простим прикладом роботи rootkit є перезапис таблиці системних викликів. Для цього зловмисник достатньо дізнатися адреси sys_call_table та визначити певний відступ до функції із цієї таблиці та замінити її на свою.

Також зловмисник може змінити значення регістру SCTLR (для ARM), а саме вимкнути NX bit захист. Після цього зловмисник зможе записувати в пам’ять свій код та виконувати його. Ще одним із способів зловмисного використання є вимкнення MMU, це теж робиться за допомогою наведеного регістру. В даному випадку, це може ускладнити експлуатацію, але якщо у вразливій системі фізичні адреса відображаються прямо у віртуальні без відступів, то це дасть змогу виконати зловмисний код не турбуючись про механізми захисту пам’яті.

Було запропоноване рішення, яке дозволяє аналізувати динамічно завантажуванні модулі. Дане рішення використовує технологію довіреного середовища виконання, що дозволяє йому бути не доступним до руткітів.

Дане рішення можна використовувати для аналізу бінарного коду модулів, знаходячись в захищеному середовищі, не залишаючи ніякі артефакти, що дають змогу руткіту упізнати аналізуючу систему.

Пропоноване рішення представляє собою певний Sandbox, в якому працює модуль. Якщо модуль звертається до важливих до інформаційної безпеки ресурсів, то замість інструкцій звернення, наприклад, до регістру SCTRL встановлюється виклик команди SMC, а далі обробник виключень буде перевіряти наведену інструкцію.

Запропоноване рішення було порівняно з типовими методами аналізу руткітів. Результат показав, що технологія TEE має ряд переваг на типовими методами. Оскільки TEE має більш привілейований доступ до ресурсів, то вона може використовуватися як монітор, логер або аналізатор системи. Це ускладнює

приховування своєї діяльності для руткітів, оскільки вони не знатимуть, що вони аналізуються.

Недоліками даної технології є складність її реалізації. Це стосується аналізу бінарного коду динамічно завантажуваного модулю. А також платформо-залежність. Даний механізм можна лише використовувати під ARM платформу.

Запропоноване рішення може використовуватися для впровадження політик роботи модулів ядра, а також може використовуватися в якості аналізатору зловмисних застосунків.

Результатом проведеного аналізу та оцінки ризиків, було виявлено, що даний проект має можливість для ринкової комерціалізації. Для представленого рішення є високий рівень попиту, що пов’язаний із станом інформаційної безпеки користувачів, тому робота над проектом має гарну рентабельність на ринку послуг у сфері кібербезпеки.

Практична цінність роботи полягає в тому, що результати роботи можуть бути застосовані в аналізі мобільних застосунків та впровадження системи аудиту, логування для мобільних застосунків.

ПЕРЕЛІК ПОСИЛАНЬ

1 ARM. ARM1176JZF-S Technical Reference Manual [Електронний ресурс]. – 2012. – Режим доступу до ресурсу: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/ch02s12s13.html

2 OP-TEE. Open Portable Trusted Execution Environment [Електронний ресурс]. -2016. -Режим доступу до ресурсу: https://www.op-tee.org/

3 QEMU. QEMU emulator [Електронний ресурс]. -2016. - Режим доступу до ресурсу:http://www.qemu.org/

4 Yiming Jing , Ziming Zhao , Gail-Joon Ahn , and Hongxin Hu. Morpheus: Automatically Generating Heuristics to Detect Android Emulators [Електронний ресурс]. – 2014 – Режим доступу до ресурсу: https://people.cs.clemson.edu/~hongxih/papers/ACSAC2014.pdf

5 Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, Sotiris Ioannidis. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware [Електронний ресурс]. – 2015 – Режим доступу до ресурсу: http://www.syssec-project.eu/m/page-media/3/petsas_rage_against_the_virtual_machine.pdf

6 Timothy Vidas, Nicolas Christin. Evading Android Runtime Analysis via Sandbox Detection [Електронний ресурс] – 2014 – Доступ до ресурсу: https://users.ece.cmu.edu/~tvidas/papers/ASIACCS14.pdf

7 ARM. System Control Register [Електронний ресурс]. – 2012. – Режим доступу до ресурсу: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500e/BABJAHDA.html

8 ARM. MMU [Електронний ресурс]. – 2012. – Режим доступу до ресурсу: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360e/BABEEGBE.html

9 9TO5MAC. Smartphone security comparison gives Apple top billing, only two Android brands do ok [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://9to5mac.com/2018/02/28/ios-versus-android-security/

10 LINUX-AUDIT. The 101 of ELF files on Linux: Understanding and Analysis [Електронний ресурс] – 2018 – Режим доступу: https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/

11 Hector Marco-Gisbert, Ismael Ripoll-Ripoll. Exploiting Linux and PaX ASLR's weaknesses on 32- and 64-bit systems [Електронний ресурс] – 2016 – Режим доступу до ресурсу: https://cybersecurity.upv.es/solutions/aslr-ng/ASLRNG-BH-white-paper.pdf

12 Hector Marco-Gisbert, Ismael Ripoll-Ripoll. Offset2lib: bypassing full ASLR on 64bit Linux[Електронний ресурс] – 2014 – Режим доступу до ресурсу: https://cybersecurity.upv.es/attacks/offset2lib/offset2lib.html

13 }{akep.Предсказание случайности. Изучаем ASLR в Linux и GNU libc, обходим защиту адресного пространства и stack canary[Електронний доступ] – 2018 – Режим доступу до ресурсу: https://xakep.ru/2018/02/19/aslr/

14 Google Project Zero. Trust Issues: Exploiting TrustZone TEEs [Електронний ресурс] – 2017 – Режим доступу: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html

15 National Vulnerability Database. CVE-2018-8781 [Електронний ресурс] – 2018 – Режим доступу до ресурсу: https://nvd.nist.gov/vuln/detail/CVE-2018-8781

16 MWR Labs Whitepaper. Kernel Driver mmap Handler Exploitation [Електронний ресурс] – 2017 – Режим доступу: https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-mmap-exploitation-whitepaper-2017-09-18.pdf

17 Quarkslab's blog. Attacking the ARM's Trustzone[Електронний ресурс] – 2018 – Режим доступу до ресурсу: https://blog.quarkslab.com/attacking-the-arms-trustzone.html

18 Phrack. Infecting loadable kernel modules[Електронний ресурс] – 2012 – Режим доступу до ресурсу - http://phrack.org/issues/68/11.html#article

19 Michael Sikorski, Andrew Honig. Practical Malware Analysis [Електронний ресурс] – 2012 – Режим доступу до ресурсу - https://nostarch.com/malware

20 Mel Gorman. Understanding the Linux Virtual Memory Manager – 2004

21 Andrew N.Soss, Dominic Symes, Chris Wright. ARM System Developer's Guide: Designing and Optimizing System Software – 2004

22 Woflang Mauerer. Professional Linux Kernel Architecture - 2008

23 Xinyang Ge, Hayawardh Vijayakumar, Trent Jaeger. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture – 2014

24 Zhenyu Ning, Fengwei Zhang. Ninja: Towards Transparent Tracing and Debugging on ARM – 2014

25 Linux Kernel. Kernel Probes (Kprobes)[Електронний ресурс] – Режим доступу до ресурсу: https://www.kernel.org/doc/Documentation/kprobes.txt

26 Ksplice. Patch Kernel Without Rebooting[Електронний ресурс] – Режим доступу до ресурсу: https://github.com/jirislaby/ksplice/blob/master/kmodsrc/ksplice.c#L1178

27 Capstone Framework. The Ultimate Disassembler [Електронний ресурс] – Режим доступу до ресурсу: http://www.capstone-engine.org/

28 Phrack. Armouring the ELF: Binary encryption on the UNIX platform [Електронний ресурс]. Режим доступу до ресурсу - http://phrack.org/issues/58/5.html

29 Statista. Number of Smartphone users [Електронний ресурс]. Режим доступу до ресурсу: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/